RADC-TR-89-259, Vol VI (of twelve)
Interim Report
October 1989

NORTHEAST ARTIFICIAL INTELLIGENCE CONSORTIUM ANNUAL REPORT - 1988 Building an Intelligent Assistant: The Acquisition, Integration and Maintenance of Complex Distributed Tasks

Syracuse University

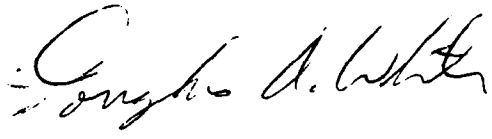
Victor R. Lesser, W. Bruce Croft, Beverly Woolf

This effort was funded partially by the Laboratory Director's fund.

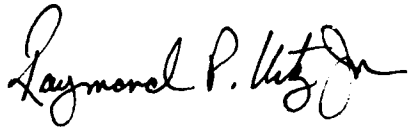
ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

This report has been reviewed by the RADC Public Affairs Division (PA) and is releasable to the National Technical Information Services (NTIS) At NTIS it will be releasable to the general public, including foreign nations.

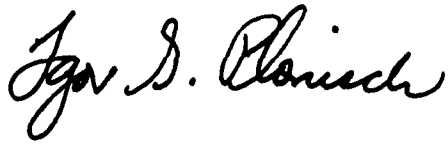
RADC-TR-89-259, Vol VI (of twelve) has been reviewed and is approved for publication.

APPROVED: 

DOUGLAS A. WHITE
Project Engineer

APPROVED: 

RAYMOND P. URTZ, JR.
Technical Director
Directorate of Command & Control

FOR THE COMMANDER: 

IGOR G. PLONISCH
Directorate of Plans & Programs

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COES) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS N/A
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) N/A	5. MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-89-259, Vol VI (of twelve)
6a. NAME OF PERFORMING ORGANIZATION Northeast Artificial Intelligence Consortium (NAIC) 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Rome Air Development Center (COES)
6c. ADDRESS (City, State, and ZIP Code) Science & Technology Center, Rm 2-296 111 College Place, Syracuse University Syracuse NY 13244-4100	7b. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Rome Air Development Center 8b. OFFICE SYMBOL (If applicable) COES	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F30602-85-C-0008
8c. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO.: 62702F; PROJECT NO.: 5581; TASK NO: 27; WORK UNIT ACCESSION NO: 13

11. TITLE (Include Security Classification)
NORTHEAST ARTIFICIAL INTELLIGENCE CONSORTIUM ANNUAL REPORT - 1988 Building an Intelligent Assistant: The Acquisition, Integration, and Maintenance of Complex Distributed Tasks

12. PERSONAL AUTHOR(S)
Victor R. Lesser, W. Bruce Croft, Beverly Woolf

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Interim	FROM Jan 88 TO Dec 88	October 1989	184

16. SUPPLEMENTARY NOTATION This effort was funded partially by the Laboratory Directors' Fund. This effort was performed as a subcontract by the University of Massachusetts at Amherst to Syracuse University, Office of Sponsored Programs.

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP
12	05	

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Artificial Intelligence; Intelligent Interfaces; Planning; Plan Recognition; Intelligent Computer-Aided Instruction

19. ABSTRACT (Continue on reverse if necessary and identify by block number)
The Northeast Artificial Intelligence Consortium (NAIC) was created by the Air Force Systems Command, Rome Air Development Center, and the Office of Scientific Research. Its purpose is to conduct pertinent research in artificial intelligence and to perform activities ancillary to this research. This report describes progress that has been made in the fourth year of the existence of the NAIC on the technical research tasks undertaken at the member universities. The topics covered in general are: versatile expert system for equipment maintenance, distributed AI for communications system control, automatic photointerpretation, time-oriented problem solving, speech understanding systems, knowledge base maintenance, hardware architectures for very large systems, knowledge-based reasoning and planning, and a knowledge acquisition, assistance, and explanation system.

The specific topic for this volume is the development of intelligent interfaces to support cooperating computer users in their interactions with a computer.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Douglas A. White	22b. TELEPHONE (Include Area Code) (315) 330-3564
	22c. OFFICE SYMBOL RADC (COES)

DD Form 1473, JUN 86 Previous editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

Item 10. SOURCE OF FUNDING NUMBERS (Continued)

Program Element Number	Project Number	Task Number	Work Unit Number
62702F	5581	27	23
61102F	2304	J5	01
61102F	2304	J5	15
33126F	2155	02	10
61101F	LDFP	27	01

Volume 6

1988 ANNUAL REPORT
To Rome Air Development Center

Building an Intelligent Assistant:
The Acquisition, Integration, and Maintenance
of Complex Distributed Tasks

Victor R. Lesser
W. Bruce Croft
Beverly Woolf

Department of Computer and Information Science
University of Massachusetts
Amherst, Massachusetts

Contents

6.1 Executive Summary

The NAIC Research Group at the University of Massachusetts has made progress in the development of systems that provide support to multiple users in the accomplishment of tasks on the computer. This work has led to the development of several planning and plan recognition systems, as well as applications in several new domains. The systems provide diverse functionality, including the ability to reason about both static and dynamic aspects of environments. The systems include: a planner that understands (recognizes) the goals of its user, achieves several goals, and relates these goals to other users' goals; a planner that formulates interactive plans to accomplish goals, monitor task execution, and resolve exceptions; and another system that resolves conflicts through negotiation.

The GRAPPLE system monitors a user's activities, detects errors, and reasons about the user's plans. It uses domain knowledge to make plausible assumptions about missing values in an open world application. Imprecise knowledge is used to support conjecture conclusions and to provide improved error detection, prediction, and disambiguation.

POLYMER is a planning system which constructs partial plans and executes them interactively. It uses constraints from agent actions to extend its partial plans. Exception handling is achieved by classifying exceptions and constructing an explanation of how each action may fit into the current plan. Plan acquisition is supported by a graphical interface based on how people recall their tasks.

We have begun a new project to look at planning in dynamic domains. This work uses simulation techniques in conjunction with classical planning techniques to solve a wider class of planning problems.

Negotiation between agents involved in conflicting or inconsistent long-term knowledge plans is handled through yet another system which initiates negotiation, assists the negotiation process, and uses both compromises and integrative bargaining to reach a solution.

We have also built a suite of programming tools that enables authors to browse and explain knowledge in an expert system for tutoring. These tools facilitate tracing and summarizing the reasoning within an expert system and allow an author to interactively modify system reasoning and response in an intelligent discourse system.

In sum, our research this year has focused on issues of inferring and anticipating goals, communicating and comprehending multiple users, and on constructing approachable and informative interfaces. As a result of this research program alone, we expect to graduate nearly 5 Ph.D. students in the next year and a half.

6.2 Ancillary Activities

6.2.1 Degrees Conferred

- Ph.D.s: 1
- MSs: 2

6.2.2 New AI Faculty

6.2.3 New AI Courses

- 689: Machine Learning
- 691B: Robotics
- 791E: Advanced Topics in Computer Vision

6.2.4 Journals, Book Chapters, and Conference Papers

Broverman, C., "Plausible Explanations to Cope with Unanticipated Behavior in Planning," Department of Computer and Information Science, University of Massachusetts at Amherst, Technical Report 88-56.

Conry, S.E., Meyer, R.A. and Lesser, V.R. "Multistage Negotiation in Distributed Planning," in *Readings in Distributed Artificial Intelligence*, A. Bond and L. Gasser (eds.), Morgan Kaufmann Publishers, pp. 367–384, 1988.

Croft, W.B. and Lefkowitz, L.S. "Knowledge-based Support for Cooperative Activities," *Proceedings of HICCS-21*, pp. 312-318. (Also in *Readings on Distributed Artificial Intelligence*, Bond and Gasser (eds.), Morgan Kaufmann Publishers, pp. 599–605, 1988.

Croft, W.B. and Lefkowitz, L.S. "Using a Planner to Support Office Work," *Proceedings of the ACM Conference on Office Information Systems*, pp. 55–62, 1988.

Croft, W.B. and Lefkowitz, L.S. "A Goal-based Representation of Office Work," *Office Knowledge: Representation, Management, and Utilization*, W. Lamersdorf (ed.), Elsevier Science Publishers, pp. 99–124, 1988.

Eliot, C., "A Survey of Recent Planning Research," Department of Computer and Information Science, University of Massachusetts/Amherst, Master's Project, February 1988.

Huff, K. and Lesser, V.R., "A Plan-based Intelligent Assistant That Supports the Software Development Process," accepted for publication in *The Third ACM Symposium on Software Development Environment*, Boston, November 1988.

Huff, K. and Lesser, V.R., "Plan Recognition in Open Worlds," Department of Computer and Information Science, University of Massachusetts/Amherst, Technical Report 88–18, March 1988.

Johnson, P.M., "Inferring Software System Structure," Department of Computer and Information Science, University of Massachusetts/Amherst, Technical Report 88-37, May 1988.

Johnson, P., Hildum, D., Kaplan, A., Kay, C., and Wileden, J., "Ada Restructuring Assistant," in *Proceedings of the Fourth Annual Conference on Artificial Intelligence and Ada*, Fairfax, Virginia, November 1988.

Lander, S., and Lesser, V.R, "Negotiation Among Cooperating Experts," in *Proceedings of the 1988 Workshop on Distributed Artificial Intelligence*, Lake Arrowhead, California, May 1988.

Lander, S. and Lesser, V.R. "Negotiation to Resolve Conflicts Among Design Experts," in *Proceedings of the 1988 Workshop on AI in Design*, held in conjunction with AAAI-88, St. Paul, Minn., August 1988.

Lefkowitz, L.S. and Lesser, V.R. "Knowledge Acquisition as Knowledge Assimilation," (short version) in *Proceedings of the Second AAAI Knowledge Acquisition for Knowledge - Based Systems Workshop*, Banff, Canada, pp. 14.0–14.14, October 1987.

Mahling, D., and Croft, W.B., "Relating Human Knowledge of Tasks to the Requirements of Plan Libraries," to appear in *International Journal of Man-Machine Studies*, 1988.

Mahling, D. and Croft, W.B., "Knowledge Acquisition for Planners," *Proceedings of the Third Knowledge Acquisition for Knowledge-based Systems Workshop*, 1988.

Mahling, D. and Croft, W.B., "An Interface for the Specification of Office Activities," in *Proceedings of the IFIP WG8.4 Conference*, pp. 233–248, 1988.

Nirenburg, S. and Lesser, V.R. "Providing Intelligent Assistance in Distributed Office Environments," in *Readings in Distributed Artificial Intelligence*, A. Bond and L. Gasser (eds.), Morgan Kaufmann Publishers, pp. 590–598, 1988.

Slovin, T. and Woolf, B., "A Counselor Tutor for Personal Development," *Proceedings of the International Conference on Intelligent Tutoring Systems*, University of Montreal, Montreal, Canada, 1988.

Woolf, B., "Intelligent Tutoring Systems: A Survey," published in Shrobe, H. and the American Association for Artificial Intelligence (eds.), *Exploring Artificial Intelligence*, Morgan Kaufmann Publishers, Palo Alto, CA, 1988.

Woolf, B., Suthers, D., and Murray, T., "Discourse Control for Tutoring: Case Studies in Example Generation," to be published as a COINS Tech Report, University of Massachusetts, Amherst, MA.

Woolf, B., "Intelligent Tutoring Systems and Multimedia Communication Systems," in *International Symposium of Computer World '88*, Kobe, Japan, Kansai Institute of Information Systems, Osaka, Japan, 1988.

Woolf, B., "Applying Artificial Intelligence Techniques to Education," in the *Proceedings of Computers, Education, and Children*, a conference sponsored by ADCIS and the Soviet Academy of Science, USSR, 1988.

Woolf, B., Murray, T., Suthers, D., and Schultz, K., "Primitive Knowledge Units for Intelligent Tutoring Systems," *Proceedings of the International Conference on Intelligent Tutoring Systems*, University of Montreal, Montreal, Canada, 1988.

Woolf, B., "20 Years in the Trenches: What Have We Learned About Building Intelligent Tutoring Systems?" invited talk, *Proceedings of the International Conference on Intelligent Tutoring Systems*, University of Montreal, Montreal, Canada, 1988.

Woolf, B., Schultz, K., and Murray, T., "Working with Expert Teachers to Distinguish Knowledge from Theory," to be published as a COINS Technical Report, University of Massachusetts, Amherst, MA.

6.2.5 Progress or Products Resulting from NAIC Research

TECHNOLOGY TRANSFER

Karen Huff gave a talk on July 29, 1988 at Massachusetts Computer Associates, in Wakefield, MA, entitled "A Plan-based Intelligent Assistant that Supports the Software Development Process."

Victor Lesser gave an invited seminar on "Intelligent Assistants" at Digital Equipment Corporation, Marlborough, Mass., January 29, 1988.

Beverly Woolf demonstrated tutoring systems to a group of visitors from DEC on February 16, 1988 and to another group from Panasonic/Matsushita, Japan, on February 4, 1988.

Beverly Woolf presented talks about intelligent tutoring systems to Hewlett-Packard, Apple Computer, Next Inc., and Boeing Aircraft, in Seattle, WA in February 1988.

Industrial Affiliates Program During the past year, COINS has added two new members to the Industrial Affiliates Program - Kodak and Ricoh. An on-campus center is being established presently to supplement the COINS basic research program with complementary basic research driven by industrial problems. The Center for Research on Intelligent, Real-Time Computation in Complex Systems (CRICCS) will allow industrial sponsors to impact research directions and work within the COINS environment in basic research areas of particular interest. The other new initiative by COINS in the area of technology transfer is the recent incorporation of a private research institute, which is being established in the immediate proximity of the University. The Applied Computing Institute of Massachusetts Inc. (ACSIOM) will represent an extension of COINS which will conduct applications research and development under contract with

industry. ACSIOM will commercialize appropriate elements of COINS research, license software, provide consulting and training services to industry, and spin-off high-tech, start-up companies in this region of the state. ACSIOM is recruiting its own staff of post-doctoral research scientists. The COINS/CRICCS/ACSIOM formula for technology transfer represents an ideal mechanism for converting the discoveries and products of COINS research into useful and beneficial applications which will strengthen the local, state, and national competitiveness in world high technology markets.

SEMINARS/WORKSHOPS

Carol Broverman participated in the AAAI workshop on Explanation-Based Learning in Palo Alto, California (Stanford) March 22-24, 1988.

Bruce Croft participated in the NAIC Executive Committee Meeting at Syracuse University on January 25, 1988.

Karen Huff attended the Fourth International Software Process Workshop in Devon, England in April, 1988.

Karen Huff attended a KBSA Workshop at Harvard University in January 1988.

Karen Huff attended AAAI-88 and the Plan Recognition Workshop in St. Paul, Minnesota, on August 24, 1988.

Susan Lander attended the AI in Design Workshop held at AAAI-88 on August 24, 1988.

Larry Lefkowitz participated in the 2nd AAAI Knowledge Acquisition for Knowledge-based Systems Workshop in Canada, October 17-24, 1987.

Victor Lesser attended the RADC Spring Meeting in Washington, DC, March 28-29, 1988.

Victor Lesser participated in the NAIC Executive Committee Meeting at Syracuse University on October 22, 1987.

Beverly Woolf attended a meeting of the program committee for the International Conference on Intelligent Tutoring Systems, February 21-22, 1988 in Montreal, Canada.

Beverly Woolf attended the RADC Annual meeting at Minnowbrook in July, 1988.

TECHNICAL PRESENTATIONS

Edmund Durfee presented a talk on "Using Partial Global Plans to Coordinate Distributed Problem Solvers" at the NAIC Annual Fall Meeting held at Clarkson University October 1-2, 1987.

Karen Huff, at the Harvard meeting, spoke informally on our research progress: expected prototype system description, contributions of the research, and open issues/future work, in January 1988.

Susan Lander presented a paper entitled "Negotiation Among Cooperating Experts," at the DAI workshop in Lake Arrowhead, California in April 1988.

Victor Lesser presented an overview of Progress on Intelligent Assistants at the RADC Spring Meeting in Washington, DC, March 29 ,1988.

Beverly Woolf was a panelist in a television program produced by the United States Information Agency in Washington, D.C. on October 2, 1987.

Beverly Woolf presented a seminar of her research results at Information Science Institute (ISI) in Los Angeles on November 9, 1987.

Beverly Woolf presented her work to a meeting of National Science Foundation PI's in Phoenix, Arizona, January 7-8, 1988.

Beverly Woolf presented a talk about intelligent tutoring systems to the School of Education at San Francisco State University in February 1988.

Beverly Woolf gave a talk at the Industrial Affiliates Meeting at UMass on May 9, 1988 as well as two talks at Smith College on May 19 and May 20, 1988.

6.2.6 Improvements to Research Environment

We have purchased 10 Texas Instruments microexplorers during this last year. Our current environment also includes 31 Texas Instruments Explorer-II's and 11 Texas Instruments Explorer-I's, plus 6 Sun-4 workstations dedicated to AI researchers.

6.3 Overview of Research

The NAIC-affiliated AI group at the University of Massachusetts has developed several interfaces that support cooperating computer users in their interactions with a computer. These interfaces contain knowledge about typical methods used by people to achieve tasks and knowledge about how to recognize a user's plans. The systems help users complete their task and provide explanations in support of their activities. We describe this work in terms of this year's accomplishments in the areas of planning and plan recognition, knowledge acquisition and display, reasoning about plan recognition, tutoring systems, and cooperative problem solving.

6.3.1 Planning and Plan Recognition

Plan Recognition We use plan recognition to help provide intelligent assistance to persons carrying out complex tasks on a computer. For example, GRAPPLE provides the context within which the machine can understand a user's activities. It is a plan recognition formalism that provides a hierarchy of procedural descriptions or plans specifying typical user tasks, goals, and sequences of actions to accomplish goals. Included within this formalism is a framework for meta-plans and first principles knowledge. This formalism has been applied to the development of an intelligent assistant that supports a programmer in carrying out the process of software development.

The success of a plan-based approach depends on capturing knowledge of the user's complex process within the planning formalism. For example, in the domain of software development, we have identified a user's goals, how individual tools are used, when special cases arise, how to recover from different types of failures, and which first-principles knowledge is relevant. The programming process laws we use for representing the likely sequence of actions that the user will perform in order to achieve a goal are captured in axioms that apply to an extended domain state, and in default rules that compensate for the fact that the extended state may be incomplete.

We have shown that two techniques, meta-plans and non-monotonic reasoning, can be used to significantly extend the representational power of the system. For example, the explicit expression of programming process laws allows reasoning from *first principles.* That is, as an independent issue, no set of rules specifically addresses which software system a user will choose as a baseline. Rather, first-principle rules relate each choice to a deeper model involving specifications, and additional rules that allow reasoning within this deeper model. The nature of the default rules determines the degree of certainty in this reasoning; with a suitable set of default rules, the interface will occasionally draw faulty (but correctable) conclusions, as a result of reasoning independently from the programmer with incomplete information.

GRAPPLE makes transformations on operations of some world state; in this case, the world state is the plan network. Transformations can be formalized as meta-operators and synthesized into meta-plans. This approach adds to the role of meta-plans, which have been used to implement control strategies and to capture domain-independent knowledge.

The primary advantage of meta-plans is the power of having expressive generality in a single formalism, as compared with a collection of ad hoc operator language extensions for each encountered exception. Any aspect of an operator definition (such as preconditions, subgoals, constraints, or effects), as well as any aspect of an operator instance (such as bindings of variables or ancestor operator instances) can be accessed or modified. The transformational approach also addresses practical problems in providing a complete library of operators. Because knowledge of exceptions is partitioned from knowledge of normal cases, the two issues can be tackled separately. The process of writing operators is further improved because multiple transformations can apply to a single operator, thereby preventing combinatorial explosion in the numbers of operators. (See Appendix 6-A)

Reasoning about Plan Recognition Plan recognition, often called interpretation, is a complex and uncertain process which requires sophisticated evidential reasoning and control schemes. We have developed a framework which models interpretation as a process of gathering evidence to manage uncertainty. The key components of the approach are a specialized evidential representation system and a control planner with heuristic focusing. The evidential representation scheme includes explicit, symbolic encodings of the sources of uncertainty in the evidence for the hypotheses. This knowledge is used by a control planner to identify and develop strategies for resolving uncertainty in the interpretations.

Since multiple, alternative strategies may be able to satisfy goals, the control process must involve search. Heuristic focusing is applied in parallel with the planning process in order to select the strategies to pursue which will control the search. The focusing scheme is flexible, allowing control to switch back and forth between strategies, depending on the nature of the developing plans and changes in the domain.

The model of evidence that we use is based on the requirements for control (i.e., that the *sources of uncertainty* for each hypothesis can be used to drive the control process).

This work is an example of the utility of making control decisions through planning. It expands on existing research work in control knowledge for planning in three significant ways: the control task is viewed as being driven by the need to resolve uncertainty, uncertainty of the interpretation hypotheses is represented explicitly and symbolically, and the process of finding the correct control plan may be seen to involve a search process which requires focusing.

We presently have a prototype implementation of this framework which simulates a system for monitoring aircraft. A variety of data sources such as acoustic sensors, radar, and emissions detectors are included. Additional sources of evidence such as terrain, air defense positions, and weather information are also available. Active control over evidence gathering can be effected through control of the operations of some of the sensors. Interpretation hypotheses cover a variety of missions including those involving coordination of multiple aircraft. (See Appendix 6-B)

Planning and Execution of Tasks A second, larger system we have developed is POLYMER, an interactive planner designed to assist in the management of tasks in a cooperative setting. The system assists in the performance of multiagent, loosely-structured, underspecified tasks.

POLYMER models an application domain through the description of *activities*, *objects* and *agents* within the domain. The representation includes both the *goal* and *preconditions* of an activity as well as a *decomposition* of the activity into smaller steps. The steps are usually goals for which other activities will be selected during the planning process, but may also be specific activities or tool invocations (i.e., "actions"). Causal relations between steps may be specified and these relations are used to generate temporal ordering constraints as well as protection intervals.

Using these domain descriptions, POLYMER interactively generates hierarchical, partially-ordered plans to accomplish a stated goal. The planning process is unique in its use of a

combination of "script based" and "goal directed" descriptions of activities to overcome the rigidity of scripts while greatly reducing the cost of purely goal-driven systems. It interleaves planning and plan execution in order to overcome the ambiguity inherent in complex, underspecified domains.

POLYMER uses a hierarchical planner to construct a procedural net that specifies the sequences of actions required to achieve a goal. It constructs partial plans and executes them in cooperation with agents. The actual actions taken by these agents are compared to expected actions, and when differences are found (producing an exception) SPANDEX is invoked (see next section).

The process of *planning* is viewed as iterative *transformations* on plan networks. A complete *plan* is a plan network which has been fully ordered, and every node is either a *phantom*[1] or it is a primitive activity node that has already been executed. Thus, a plan network represents a class of complete plans; there are multiple possible complete plans that may result depending on the choices of elaborations and operations that are subsequently applied. As a plan network is further elaborated and executed by the plan network maintenance system, a *plan history* is built up.

The POLYMER planning system has been designed and implemented as the core of an environment to support cooperative work. The current prototype has been used to interactively generate plans in such diverse areas as journal editing, software development and house purchasing. In addition to the development of further applications, POLYMER is now serving as a testbed for several other research projects. These projects are exploring the use of knowledge acquisition, exception handling, and computer-mediated conflict resolution as part of an effort to develop an integrated environment for the support of cooperative work. (See Appendix 6-C)

Plausible Reasoning about Exception Handling No matter how carefully a plan is conceived, things frequently go wrong during its execution. When human agents are responsible for executing plan steps this problem is of particular importance. People change their minds or opportunistically revise a plan mid-execution. In addition, people are prone to error and misjudgment. Consequently, a system designed to support the performance of human tasks in an interactive setting must be prepared to cope with frequent unanticipated occurrences (*exceptions*) during the planning and execution process. For example, an agent may leave out a step in a task as a deliberate short cut, or he may perform an unexpected action as an intentional substitution of an expected action. In other cases, the action of an agent may not be an intentional aberration, but may be viewed initially as an exception due to an incomplete or incorrect domain plan library. Such exceptions are referred to as *accountable* since it is presumed that *agents behave purposefully* and there are *motivations*.

SPANDEX recognizes actions which initially appear to be "errors" and then tries to explain them as actions consistent with the goals of the plan. This class of accountable exceptions is contrasted with the more frequently addressed arbitrary changes in a world

[1] A phantom node is a goal node which has been determined to be true at its position in the plan without further expansion and execution.

state brought about by unknown agents. *Accountable* exceptions should be justified rather than "counteracted" through replanning. Explanations of unanticipated agent behavior can result in improvements in both system understanding and performance. As an initial step towards achieving this aim, we have defined the categories of accountable exceptions that can arise in a cooperative planning framework. The types of exceptions defined by this behavioral perspective are: *action not in plan, out of order action, repeated action, user assertion*, and *expected action with parameter causing constraint violation.*

SPANDEX allows a planning system to continue the planning and execution of a task after encountering an exception. The approach facilitates extension of an existing knowledge base to accommodate alternative ways to complete task goals, as learned through the handling of previous exceptions. The *replanner* handles unaccountable exceptions.

We have tested SPANDEX in domains used to support the cooperative work of one or more human agents. In such environments, human input is used to guide the development of a plan for a task. We use human-generated plan exceptions which are incorporated into the developing plan. Alternatively, other systems use replanning techniques which generally "undoes" the original plan. (See Appendix 6-D)

Planning in Dynamic Environments Classical planners cannot reason about complex dependent effects in their environment and are limited in that they make strong assumptions which are often violated in the real world. In particular, it is generally assumed that the only cause of changes is the actions planned by the system. Domains in which the world changes state due to causes not directly under the control of the planner are *dynamic domains.* Some things we can control indirectly. For example, a kettle of water will heat to a boil when it is placed on a hot stove, and it will cool down when the stove is turned off. The temperature of the water is not under our direct control, but the burner of the stove is, so we can make the water boil indirectly. Other things cannot be controlled, but often we can make predictions. For example, the phase of the moon cannot be controlled but it can be predicted with great precision.

We use simulation techniques in conjunction with classical planning techniques to build a system that can solve a wider class of planning problems, including those involving dynamic domains. We are interested in domains where the planner is not entirely in control but is generally able to make predictions.

A simple change to the blocks world example illustrates planning in a dynamic domain. Suppose that three blocks lying on a table are to be stacked in a column and that every action requires a unit amount of time. Also suppose the blocks change color in a blinking fashion: At the end of every unit time interval all shaded blocks become white, and all of the white blocks become shaded. The color of all blocks is specified in the initial state and the goal. The stacking problem can still be solved, but may require the addition of an 'idle' action to correct the color of the blocks. Of course some problems in this domain will be unsolvable because the parity of the blocks is wrong.

Plans in dynamic domains are originally derived from plans generated using classical techniques. Once the static domain constraints are satisfied by a candidate plan, the plan

is further analyzed using a detailed simulation model. The simulation produces a causal analysis of the interaction between the plan and its environment. Dynamic behavior is never represented in the plan, but only the constraints implied by it. These constraints are derived from the simulation history and used to correct the initial plan.

The simulation language is an important part of the system. We have chosen to implement a process model discrete event simulator. This type of simulation is efficient, well understood and complements the strengths of standard planners. Existing planners provide goal-based, categorical reasoning about static events. Simulation provides statistical reasoning about dynamic events. It also provides answers to *what if* questions about future states of the world. Combining the different reasoning methods allows more domain constraints to be represented and applied to the planning problem.

An assumption of this approach is that plans generated using a static domain description will be good approximations of plans that work in dynamic domains. Once a plan satisfies the static constraints, it is simulated to determine if it also satisfies the dynamic domain constraints. If so, the system can increase its confidence in the plan. Otherwise the plan is discarded or modified to avoid the particular problems found.

A plan does not have to be complete before it is simulated. Unexpanded subgoals in an incomplete plan are treated as operators that instantly produce the desired effect. By simulating a plan before it is complete the system can uncover defects earlier. The optimal tradeoff between simulation and planning is a control issue that has not been addressed.

We are using statistical information and context dependent information derived from the simulation knowledge to substantially enhance the ability of the planner to handle dynamic environments. (See Appendix 6-E)

Recognition of Complex Processes Structural evolution is a particularly difficult problem in software development. AI software is particularly susceptible to evolution, since it is both ***experimental*** and ***domain-embedded.*** Experimental software is used to discover problem solutions, not simply to reify in computable form a solution discovered through pre-implementation analysis. Domain-embedded software is software that, by its existence, changes the nature of the problem to be solved. Structural evolution is poorly supported by traditional development methods and environments, which assume the behavior and structure of software can be tightly constrained before implementation.

Recognition, representation, and support for structural evolution is thus an important component of development methods and environments for AI software. We are exploring how knowledge about structural evolution can be computationally encoded and intelligently applied during the development process.

We believe that representation of software structure at *multiple grain sizes* offers significant potential for addressing key problems in evolving software systems. In general terms, our research views software structure at the following grain sizes: (1) a collection of ascii characters; (2) a set of functions, procedures, and variables; (3) a set of objects and/or abstract data types; (4) a system; and (5) as a member of a community of co-evolving software systems. Traditional environments offer separate, relatively un-

integrated tools for expressing structure at each grain size. In contrast, our research is exploring tools to represent and support evolutionary *processes* at each grain size, as well as the impact evolution at one grain size has on other grain sizes.

Through appropriate representation of *knowledge about software structure* evolution, we hope to address three significant problems in evolutionary software:

Overhead of structural evolution. Structural reorganization is time-consuming and error-prone when done by hand. This environment provides automated support for many types of structural change, catalyzing the evolutionary process.

Access to software resources. Separate development projects often require subsystems (such as graphic interfaces) within similar functionality. The different structural contexts required for these common subsystems often impede their re-use. Representation of the *community level of software*, and *structural processes* at this and lower levels can support the structural reorganization required to more efficiently and completely exploit community resources.

Development team coordination. In traditional development, the original structure is designed to allow concurrent and autonomous subsystem development. In evolutionary contexts, the extant software structure may not support this asynchrony. These environments can help in reorganizing the system structure so that developers can add and modify behavior without "stepping on each other's toes."

(See Appendix 6-F)

6.3.2 Knowledge Acquisition and Display

Knowledge Acquisition and Display of Plans A critical obstacle to the use of knowledge-based systems, such as rule-based expert systems and planners, is the difficulty in acquiring knowledge about the user's environment. To make planners more accessible to end users we have developed an interface that makes the functionality of the planner seem more natural and immediately understandable to a user. In particular, we have designed a graphical interface for the acquisition, display and debugging of plan knowledge.

DACRON, a graphical planning interface supports the acquisition of plan knowledge by providing representations from the user's point of view. It is based on a model that relates the elicitable knowledge of the end-user, who is the domain expert, to the requirements of the planner.

The model makes predictions about what humans know about tasks. To check the validity of the model, we have conducted several experiments with more than 150 subjects. In these studies, operations, which are the lowest level of description of the task, were grouped by the domain experts into units. A task is an activity which is viewed by the human as a unit at its standard level of abstraction (e.g., *purchase an item* for a secretary or *receive reimbursement for travel* for a manager). The grouping of operators

makes statements about structures and processes involved in the recall of complex tasks from long term memory. The act is the smallest coherent unit in the description of a task that appears to be at the appropriate level of abstraction.

To acquire plans, the stages of knowledge acquisition were reviewed with the domain experts, including explaining to them what was to be acquired, and deriving from them a way to elicit, code, display, and debug the resultant knowledge. In order to build the plan acquisition system, a model was generated that links the elicitable knowledge of the domain experts to the requirements of the planner. Having such a model allows us to design the plan acquisition system with some confidence that domain experts will be able to use it.

Previous systems in knowledge acquisition have primarily addressed rule-based expert systems and concept hierarchies in knowledge bases. The special constraints in the acquisition of knowledge for planners have not been explicitly identified. In this work, the requirements that situation calculus-based planners pose on the knowledge acquisition process have been uniquely identified.

DACRON describes how to purchase an item and specifies causal dependencies, responsible agents, constraints, and affected objects as well as the goal and precondition information. The interface provides icons for acts, relations and objects. Each icon can be retrieved, edited, copied, completed or debugged. Every icon can be opened with the mouse and presents the user with a graphic form editor for the particular entity that is represented by that icon. In the case of acts, the editor shows compartments that refer to the pre-situation, the operations and the post-situation of particular act, which should correspond to the same entities in the cognitive framework. In the case of knowledge acquisition, these compartments are ready to accept other icons as input. In the case of displays, icons representing values appear in these compartments. Constraints between items in the knowledge base are placed in the compartment that contains the object to which a certain constraint applies. In the case of inter-object constraints which might apply to objects in different compartments the users might choose the more convenient one. (See Appendix 6-G)

Knowledge Acquisition in a Discourse System We have developed mechanisms that adjust both the nature of the discourse and the content to be communicated between a machine and user based on real time changes in the discourse and the model of the user. This mechanism, called a DACTN *Discourse ACtion Transition Network*,[2] represents and controls the system's dialog with a user. It is constrained by the system's assessment of the user, a record of the user's state of knowledge, and system history.

A DACTN incorporates a taxonomy of frequently observed discourse sequences and provides a default response for the discourse system. This year, we have modified the system to reason about local context, where local context is an aggregate of the user model and response history.

The system represents the space of possible discourse situations by arcs, defined

[2]Rhymes with ACT-IN

as predicate sets, that track the state of the conversation. Nodes provide actions for the tutor. The discourse manager first accesses the situation indicated by the arcs, resolving any conflicts between multiply-satisfied predicate sets, and then initiates the action indicated by the node at the termination of the satisfied arc.

Because the system provides a structured framework for representing dialogs, a visual dialog editor has been developed, which allows an expert to create new interventions graphically and have them automatically translated into LISP code. This allows the experts to contribute to knowledge acquisition of the discourse system without having to work (excessively) with knowledge engineers. New machine interventions can continue to be elicited from experts even as development and evaluation of the system proceeds. By adding interventions to the library and by linking them to the domain knowledge we expand the system's repertoire even as we test the system.

The domain expert adds a new question or statement and is led through a series of prompts designed to elicit the possible client responses. Each response has associated with it two pieces of information: a classification of the response, which is based on the current user profile, and the profile updates related to the choice of this response. The profile modifications may include both updates based on the classification of the response and updates specific to this question and response.

As each question is added, the visual editor graph is updated so the expert always has a view of the current state of the intervention. The underlying DACTN is created dynamically so that at any point in the editing it can be executed against default profiles, allowing the expert to check the appropriateness of the machine's responses.

This on-line visual editor enables piecewise development and evaluation of the system and thus a wider circle of computer-naive authors (e.g., psychologists, teachers, curriculum developers, etc.) can participate in the process of system development. (See Appendix 6-H)

Control Systems for Explanation We have developed ways to control explanation from complex and epistemologically adequate knowledge bases. The goal is to take any knowledge base and, without necessarily knowing whether it is a semantic net or frame based, extract from it a particular representation which is adequate for explanation. In order to specify the semantic categories of the attributes of any object in the knowledge base, the control mechanism will first have to generalize those parameters needed to fully specify the explanation to be returned and then express both the object and its relation to other objects in the knowledge base.

We have designed a View Retriever mechanism which describes in epistemological terms the kind of knowledge that might be of most interest to the user and how it should be organized. Ideally, the mechanism will return a focused, perspicuous representation appropriate for each task at hand. Interesting uses of this approach to explanation include both explaining the reasoning of a problems solver as well as explaining and teaching domain knowledge behind that reasoning.

For example, consider the domain which represents knowledge about electrical networks. A view mechanism, such as described above, would be able to explain how

networks operate, and why they work as they do, in addition to explaining how to solve specific problems.

The research issues here are to describe the perspectives available to the system, to have the system choose between them and then to implement the explanation.

We have developed a "generic" description of the available perspectives and have designed several mechanisms to find the "best" way to taxonomize or otherwise describe the various views. Given that we can identify the many perspectives available, we still need to know which one to use and when. Ideally, the mechanism might choose from among these perspectives according to the user's background, role with respect to the system, goals, and the discourse context, including what question may have been asked and what is being discussed.

6.3.3 Tutoring Systems

Discourse Management for Tutoring We have designed several general purpose techniques for managing discourse in an intelligent tutor. These techniques are being implemented in structures that dynamically reason about a system's response to the user. The goal is to have the system respond fluidly to the user and to coordinate its utterances in a more flexible manner than has been required for question/answer or summarization systems. The framework we have designed represents both knowledge and control information and dynamically customizes the machine's responses to the user.

Building an intelligent tutoring system requires the ability to model and reason about domain knowledge, human thinking and learning processes, and the teaching process. Our research this year has focused on the design and implementation of three mechanisms.

Knowledge Units (KUs) supply the component elements for tutoring, student, and domain modules. They express the topic to be taught, the possible tutoring responses, and the student's knowledge. We have implemented a preliminary system for describing tutorial strategies in terms of a vocabulary of primitive discourse moves. This system, called TUPITS[3] is an object-oriented representation language that provides a framework for defining primitive components of a tutorial discourse interaction. This system is used by the tutor as it reasons about its next action.

Representing the various tutorial discourse primitives in an object-oriented system allows the system to define methods and default information differently for various classes of objects of the same basic type. For instance, KUs are classified as being for factual, conceptual, or procedural information and each sub-class has a different "describe" method associated with it. Also, individual KUs can override the default methods, such as an idiosynchratic teach-prerequisite method defined for a particular KU. During the past year we have analyzed tutorial discourse to identify common primitive discourse moves and teaching strategies.

Control Structures guide movement of the system through the KUs. These structures are currently "directive," that is, they are motivated by specific instructional and diagnostic goals and thus produce a predominantly Socratic interaction. The control

[3]TUPITS (Tutorial discourse Primitives for Intelligent Tutoring Systems)

structures specify three levels of control, separately defining the topic, presentation, and response selection. We have built a knowledge representation language that facilitates representing and modifying common tutorial action patterns. A second tool facilitates the generation and manipulation of examples.

Cognitive Processing Results are encoded in the teaching knowledge of the system. They enable a system to use expectations and make inferences about a user. The more information the system has about typical user errors and potential misconceptions, the more it can base its decisions on tutoring experience and expertise. Currently our systems have a limited amount of encoded cognitive processes knowledge.

In the last year this research has moved closer towards building generic and consistent mechanisms for eliciting and encoding declarative knowledge and control knowledge for an intelligent tutor. (See Appendix 6-I)

6.3.4 Cooperative Problem Solving

Negotiation among Distributed Experts Many complex problem-solving tasks require diverse expertise to generate comprehensive solutions. There are ubiquitous examples of expert cooperation in human problem-solving tasks such as design, research, and business management. There are many benefits to expert cooperation, especially the ability to bring the knowledge of multiple disciplines to bear on a single problem. There are also difficulties however, particularly the resolution of conflicts that arise from trying to merge multiple local goals, priorities, and evaluation criteria for a common good.

The primary emphasis of our work is to describe an architecture that provides a suitable framework for the cooperating experts paradigm and to demonstrate its efficacy through a prototype implementation. In our implementation, we represent the salient features of constraints, goals, and short- and long-term knowledge that will allow for communication and conflict resolution techniques to support cooperation. Each of the "experts" in this framework is a fully functional knowledge-based system, capable of independently solving at least some portion of the problem. The systems may have mutually inconsistent or incomplete long-term knowledge, different knowledge representations, different problem-solving architectures, and different biases. It may be impossible for the systems to share information except through formal protocols because of the lack of a common language. The systems may be working on different problems and overlap in the solution space is incidental or they may be working on the same problem and have different criteria for generating and evaluating solutions. In any case, solutions must be acceptable to all agents despite their differing priorities and interests. Because the systems potentially have different internal problem-solving mechanisms, they can't accurately predict the actions or preferences of others—an agent may not even know what other agents exist in the environment or what their interests might be. Therefore, formal mechanisms must be provided to enable the generation, communication, and negotiation of mutually acceptable solutions.

Some of the issues that must be addressed in the design of a cooperating expert framework are:

1. how to represent and store knowledge to enable communication and negotiation;
2. communication protocols for sharing information in a timely manner;
3. negotiation protocols for resolving conflicts which arise due to locally conflicting goals and priorities;
4. the scheduling and coordination of conflict resolution tasks within the overall problem-solving activities of a system.

We are currently working on a design for a cooperating expert framework in the domain of kitchen design which will incorporate both existing and custom-built knowledge-based systems. This work will lead to experimentation to understand the impact of multiple experts on solution cost and quality in varying situations. (See Appendix 6-J)

Appendix 6-A

A Plan-based Intelligent Assistant That Supports the Software Development Process

Karen E. Huff and Victor R. Lesser

Abstract: We describe how an environment can be extended to support the process of software development. Our approach is based on the AI planning paradigm. Processes are formally defined hierarchically via plan operators, using multiple levels of abstraction. Plans are constructed dynamically from the operators; the sequences of actions in plans are tailored to the context of their use, and conflicts among actions are prevented. Monitoring of the development process, to detect and avert process errors, is accomplished by plan recognition; this establishes a context in which programmer-selected goals can be automated via plan generation. We also show how non-monotonic reasoning can be used to make an independent assessment of the credibility of complex process alternatives, and yet accede to the programmer's superior judgment. This extension to intelligent assistance provides deeper understanding of software processes.

This work was supported by the Air Force Systems Command, Rome Air Development Center, Griffiss Air Force Base, New York 13441-5700, and the Air Force Office of Scientific Research, Bolling AFB, DC 20332 under contract No. F30602-C-0008, supporting the Northeast Artificial Intelligence Consortium (NAIC).

1 Introduction

Environments have traditionally provided minimal support for the process of software development. Separate parts of the process are typically supported by separate tools, while global patterns of tool usage are not made explicit, and are not exploited. Thus, the environment cannot prevent a programmer from starting compilations before an appropriate context is set up, enforce a policy of regression and performance testing before a customer release, insure that new source versions are checked back into the source code control system, or guarantee that source files are deleted only after their contents have been archived or superseded. Such support would be valuable to both programmers and their managers.

Extending environments to incorporate process support requires explicit representation of the software process, showing how software development goals are mapped into sequences of environment actions (Figure 1). Typical goals (during implementation) are concerned with adding functionality to a system version, fixing bugs, adhering to various project-specific policies, and maintaining an organized on-line workspace. The actions available to achieve these goals consist of invocations of tools provided within the environment. Knowledge of a specific software process governs the mapping from goals to actions, distinguishing between appropriate sequences of actions and random ones.

This view of software processes fits the planning paradigm, an AI approach to a theory of actions. In planning [9], knowledge of a domain is expressed in operators (parameterized templates defining the possible actions of the domain) together with a state schema (a set of predicates that describe the state of the world for that domain). Goals are logical expressions composed of the state predicates. A plan is a hierarchical, partial order of operators (with bound parameters) that achieves a goal given an initial state of the world. There are two mapping algorithms: planning, where a plan is constructed given a goal and an initial state, and plan recognition, where a plan and its goal are inferred given a sequence of actions and an initial state.

When the planning paradigm is applied to software processes, operators are the vehicle for formally defining processes; plans are the data structures that represent instantiations of processes; and, assistance can be active (via planning) or passive (via plan recognition). If the programmer retains the initiative for performing the process, issuing commands exactly as at present, plan recognition can detect and avert process errors. This "kibitzing" is an automated version of a colleague watching over the shoulder of a programmer at work. Alternatively, the programmer can state a goal to be satisfied, invoking plan generation to automate achievement of the goal. Depending on the scope of the goal, plan generation may be completely automatic, or cooperative [6] (relying on interactive input from the programmer).

We call this combination of volunteered advice and cooperative automation *intelligent process assistance*—an approach to *machine mediation* of software development [2, 18] as it applies specifically to the development process. Another type of plan-based intelligent

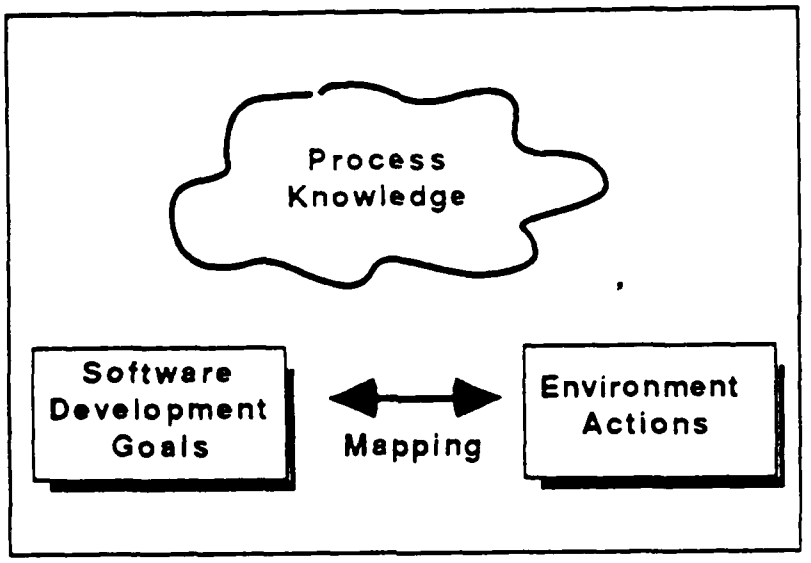


Figure 1: A View of Software Processes

assistance, directed at understanding the deep structure of code, is described in [14, 25].

Comprehensive support of the software process requires involvement in both the complex decisions as well as the mundane details. Formally representing some of these decisions is a challenge (independent of the choice of process specification formalism). What are the criteria for choosing the baseline from which to develop a new system version, selecting tests to run, or deciding which system version is releasable? If a process assistant lacks knowledge to address these decisions, it cannot independently critique a choice made by the programmer, nor can it suggest a restricted set of likely candidates from which the programmer can choose; the level of assistance is seriously restricted.

While universal decision procedures embodying these complex criteria seem unattainable, rules can be given to cover typical situations and anticipated exceptions. Reasoning with these rules is inherently imprecise; the conclusions are plausible, but fallible—assumptions may have to be revised. By introducing *non-monotonic reasoning*, it is possible to formalize additional process knowledge to evaluate the *credibility* of alternatives for decisions that could not otherwise be addressed. The additional discrimination power is flexible, not absolute; it is possible to defer gracefully to the programmer's judgment, integrating the implications of that judgment into a revised set of assumptions.

In Section 2, we show how the planning paradigm applies to software development, giving examples of software process operators and plans. Planning is compared with other approaches to specifying and supporting software processes. We also show how a

plan-based intelligent assistant is integrated into an environment architecture, and give examples of volunteered advice and automation. In Section 3, we present an approach to deeper process understanding. Additional process knowledge is expressed in monotonic and non-monotonic rules, and a *truth maintenance system* is used to reason with this knowledge. In the final section we summarize the GRAPPLE project status and describe future extensions.

2 Plan-based Process Support

2.1 Software Process Operators

As an example of a software process, consider how implementation might be carried out for a traditional programming language such as C, assuming currently accepted engineering practice such as incremental development, source code control, bug report database, and specialized test suites. A partial library of operators for this domain appears in Figure 2. A state schema supporting these operators is sketched in Figure 3, using the ER model of data [5] as the graphical presentation; relationships and attributes correspond to the logical predicates used in the operator definitions.

Operator definitions follow the state-based, hierarchical planning approach [22, 23, 28]. Each operator has a *precondition* defining the state that must hold in order for the action to be legal, and a set of *effects* that defines the state changes that result from performing the action. These core clauses are augmented by a *goal* clause that defines the principal effects of an action (thus distinguishing them from "side-effects" of the action), and a *constraints* clause that defines restrictions on parameter values. The *unit-check-in* operator in Figure 2 describes the action of checking a new version of a source module into a source code control system (such as RCS [24]). The precondition on the action requires that the module was previously checked-out with return privileges. The goal of the action is that the new version is now "under" source code control; this is also one of the effects of the action. There is one side-effect, namely that the return privilege is lost (another operator is used to describe the type of check-in that retains the return privilege).

An abstraction hierarchy is created through *complex* operators having *subgoals*. (This is essential for describing complicated processes.) The subgoals of the build operator decompose building into four parts: creation of new source versions, creation of a load module, running unit tests, and checking-in. *Primitive* operators, which do not have subgoals, correspond to the atomic actions in the domain. Some primitive actions, like *unit-check-in*, correspond to tool invocations (perhaps with specific parameter settings). Other primitive actions correspond to command-language scripts; *release* could be implemented as a script that copies a load module to the customer's directory and issues a release notice.

All the policies and procedures that are associated with a particular software devel-

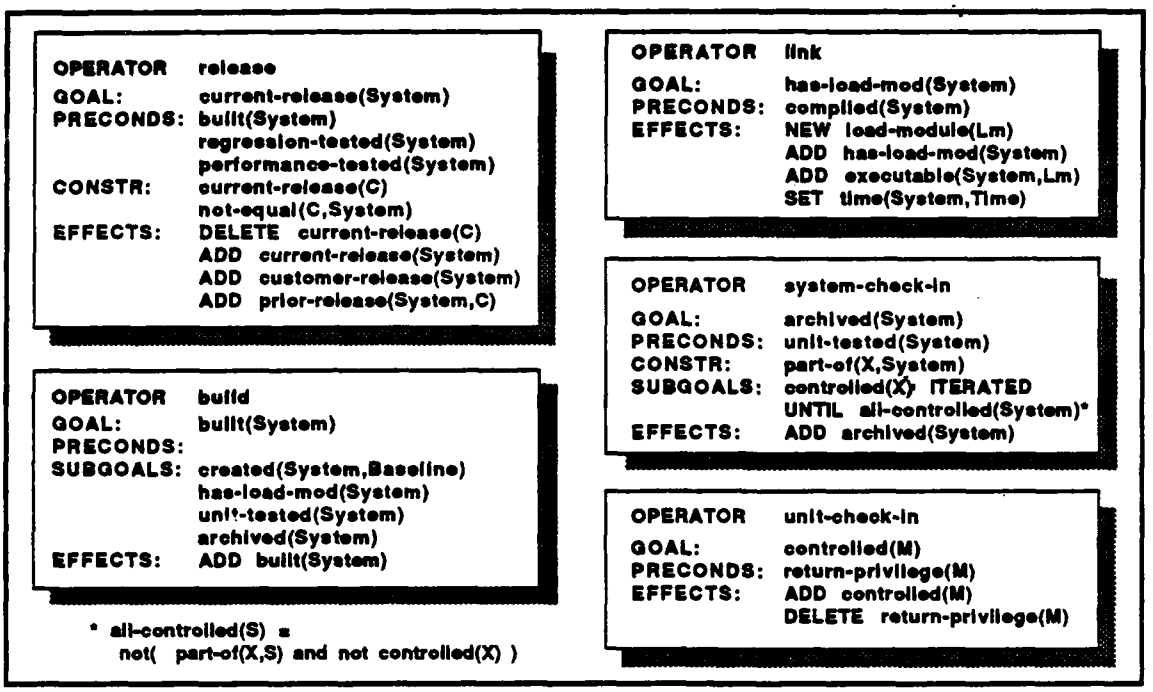


Figure 2: Software Process Operators

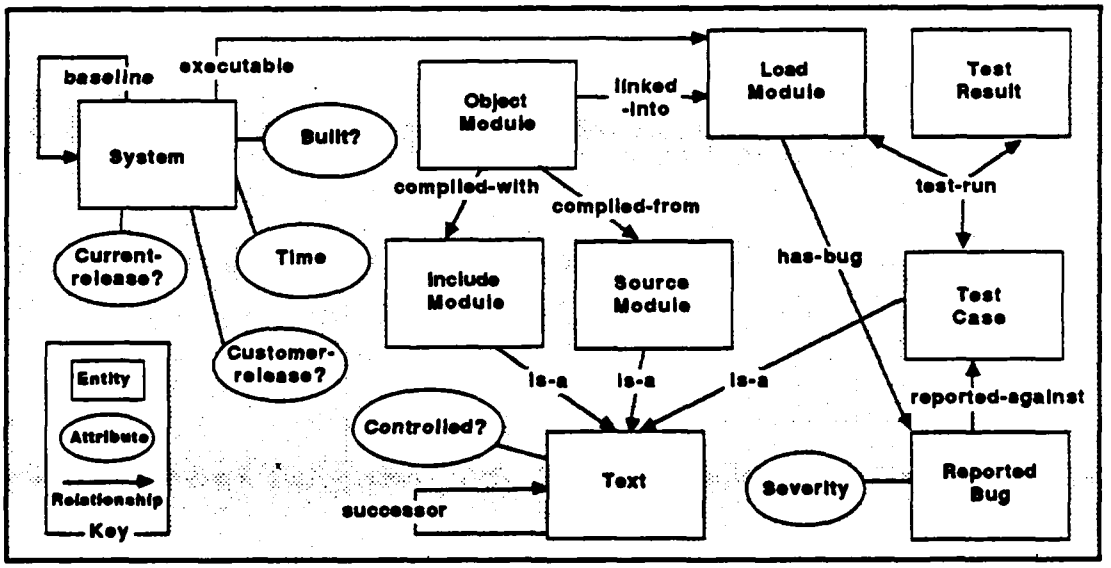


Figure 3: Software Process State

opment process must be included in the operator definitions. Changing the definitions of the operators changes the process that will be followed. For example, the *release* operator in Figure 2 requires that performance tests and regression tests be run before a customer release is made. These requirements could be relaxed (by deleting one or both testing-related preconditions) or strengthened (by adding another precondition requiring execution of a particular analysis tool or updating of release documentation).

2.2 Software Process Plans

Plans are constructed dynamically by instantiating operators. Operator definitions contain sufficient information to reason about sequences of actions without actually executing the actions. The state changes that an action causes are explicit; therefore, sequences of actions can be "simulated." The exact preconditions for an action are explicit; therefore, actions can be ordered correctly. Concurrency is implicitly allowed, subject only to the stated preconditions. Plans are automatically tailored to the exact context for which they are needed. If a subgoal has already been achieved as a side-effect of prior activity, actions to achieve that subgoal will be omitted. If part of a plan fails during execution, replanning will fill the gaps left by the failure (and only those gaps).

Planning is distinguished from other theories of actions by its emphasis on goals to achieve, not actions to perform. Determination of actions proceeds from goals, so that contingency handling (e.g., for redundancy or failures) is internal—not external—to the planning system. When process definition is procedural [19] or event-based [11], the composition and ordering of actions is predetermined; any contingency handling must be built into the definition by hand, in advance. A behavioral approach to process modeling that combines action and goal orientations is described in [29].

A partial example of a hierarchical plan is given in Figure 4. There, a vertical slice covering three hierarchical levels is shown; lower levels reveal more detail in the plan. Downward arrows between levels connect desired states with operators instantiated to achieve them; in general, there may be several alternative operators that could achieve a given state. Arrows within levels show how the achievement of certain states is partially ordered with respect to time. These temporal constraints follow from the operator definitions: precondition states must always precede subgoals, for example.

A special type of contingency arises when there are conflicts among actions. For example, consider an instantiation of the build operator that involves changing one source module. The new source module must be preserved from the time it is created until it is checked-in. An action such as editing (that would contribute to building the next system version) cannot be allowed to interfere with accomplishing the check-in. Strategies for salvaging the plan include preventing editing until check-in occurs or inserting an action to make another copy of the module. Conflicts can occur within a single plan as well as between two plans being carried out concurrently. The simple action of deleting something has the potential of interfering with almost any active plan.

Planning algorithms handle conflicts via domain-independent methods. Associated

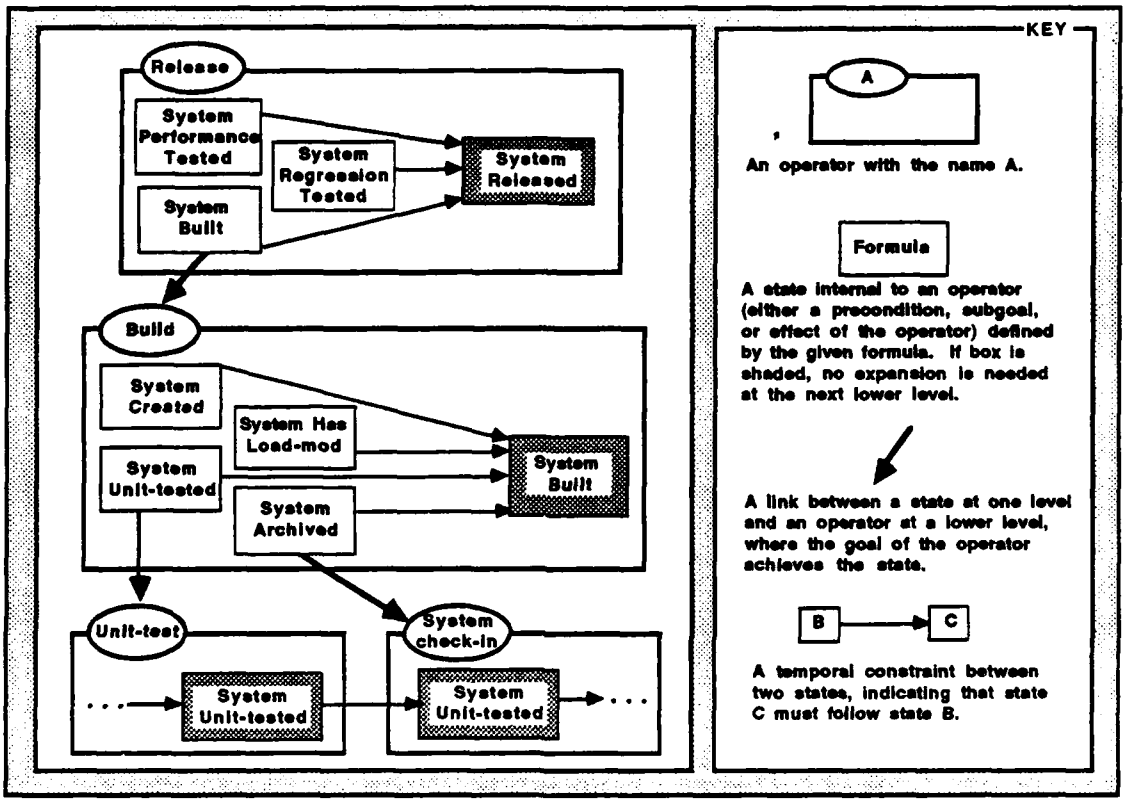


Figure 4: Vertical Slice of Hierarchical Plan

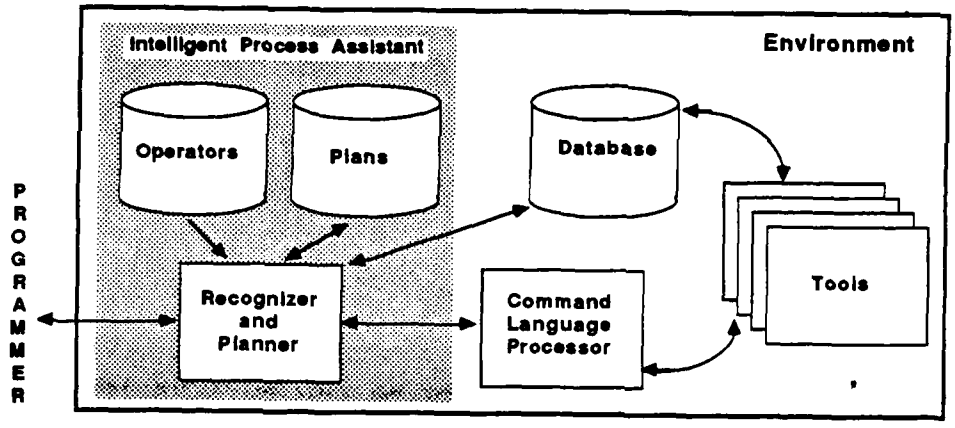


Figure 5: Environment with Intelligent Assistant

with each precondition or subgoal is a *protection interval* that begins when the precondition or subgoal is achieved and ends when the operator begins (for preconditions) or ends (for subgoals). A conflict occurs when a precondition or subgoal is destroyed during its protection interval. Domain-independent methods for resolving conflicts, described in [12, 22, 23, 28], include imposing additional temporal constraints and selecting alternative operators.

This ability to anticipate and prevent conflicts (among operators that are fundamentally *if-then* rules) distinguishes planning from other rule-based systems. Marvel [15] uses forward and backward chaining to automate chores that are prerequisites to or consequences of programmer actions. Glitter [8] cooperatively automates goals in the transformational development process. These systems do not prevent conflicts among rules. Systems based on <*condition, action*> rules, such as CLF [1] and Genesis [21], lack descriptions of effects; they cannot reason about the consequences of actions, and therefore cannot prevent conflicts. Planning techniques are used in two existing systems. Agora [4] provides a domain-specific planner for tasks relating to heterogeneous, parallel systems. Although help systems usually provide advice that is too local to qualify as process support, UC [27] now uses planning to gain a more global perspective [17].

2.3 Integration in an Environment

The integration of an intelligent assistant into an environment is shown in Figure 5. The base environment contains a command language processor that invokes the available tools. The tools in turn reference and update the environment storage system, implemented as a database [3, 10, 20] or objectbase [26], comprising not only software products but also their attributes and the relationships among them.

The intelligent assistant has an active component based on algorithms for plan recogni-

tion and plan generation. These algorithms are process independent; they obtain process knowledge by referencing a library of operators. In order to construct plans, the algorithms must reference the current state of the world, which is essentially the database already present in the environment. The intelligent assistant also contributes additional state information to this database. For example, the effects of the release operator define the "meaning" of release from a higher-level perspective than the script implementing release. The script invokes the copy command to copy the contents of one file to another; copy is not aware that a release is being copied or that the release sequence is being extended.

The programmer communicates with the intelligent assistant, which moves smoothly between passive and active assistance. For commands issued in the normal way, plan recognition is invoked to find an interpretation for the action. If a valid interpretation is found, the command is passed to the command language processor for execution. An *interpretation* is a path from the action up through the plan hierarchy to a top-level operator of an existing or new plan; an interpretation is valid if all relevant preconditions and constraints on the path are satisfied. When the programmer issues the command *achieve* <*goal*>, plan generation is invoked to produce a sequence of commands for execution by the command language processor. Plan generation can also be invoked to satisfy a precondition to make an interpretation valid. Specific examples follow.

2.4 Process Support

Consider the plan recognition situation diagrammed in Figure 6. Here, the programmer has been building a new system version (S1), and work has proceeded as far as unit-testing. Thus, three subgoals of *build* (at level 1) have been satisfied (the detail of exactly how this was accomplished has been omitted). The current state of the world is also diagrammed, showing that S1 was constructed from two source modules and one include module. Assume that the programmer has just issued a command for *unit-check-in* on module C. This action is "recognized" as fitting into the plan for building S1, because *unit-check-in* (on level 3) satisfies a subgoal in *system-check-in* (on level 2) which satisfies the remaining subgoal in *build* (on level 1). This interpretation for *unit-check-in* of C is valid, because all necessary preconditions are met: return privileges exist for C (level 3) and S1 has been unit-tested (level 2). With a valid interpretation, no advice need be volunteered.

Since the roles and interrelationships of actions are explicitly represented in plans, agendas (what needs doing) and status reports (what has been done) can be generated at multiple levels of abstraction. A plan can be constructed for any agenda item, and the rationale provided for any completed activity. In contrast, the DSEE task management facilities [16] allow users to associate actions performed with a task/subtask structure, but any intelligent processing of stored task information must be provided by the programmer.

Three additional recognition examples are:

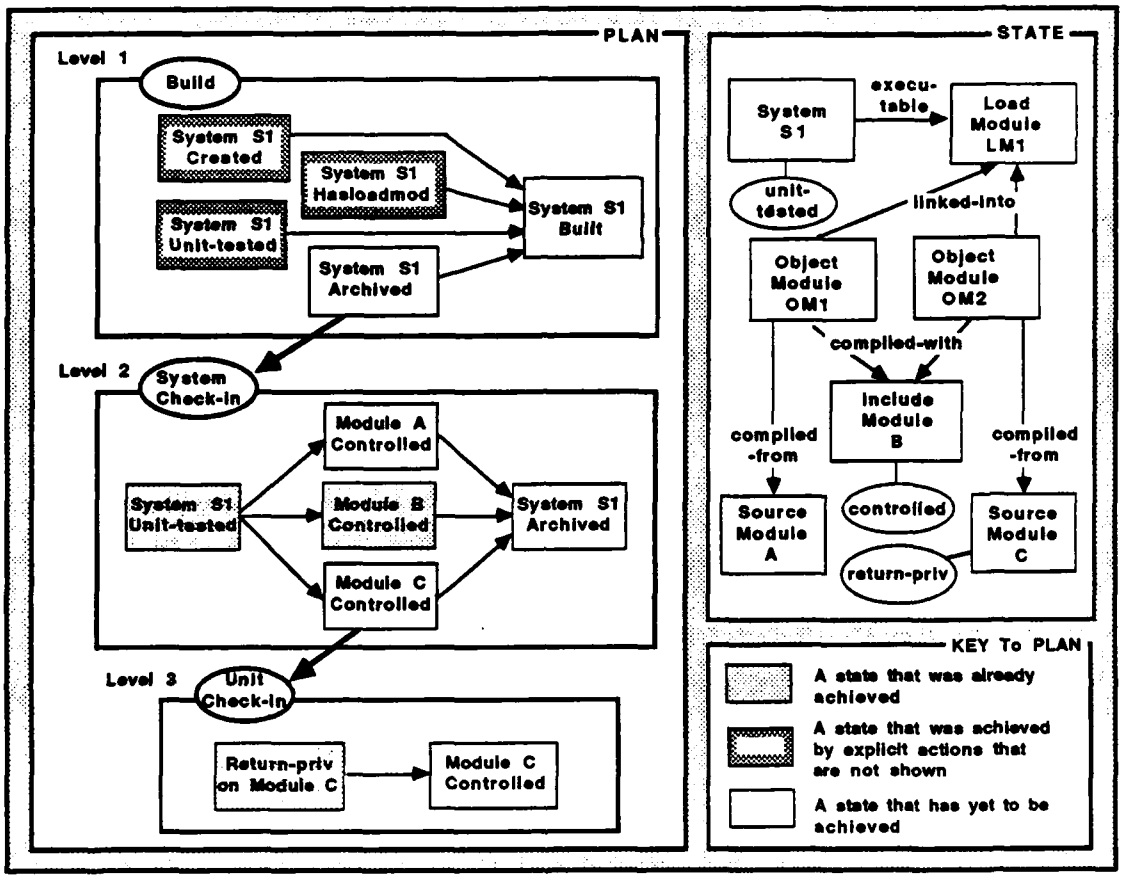


Figure 6: A Plan in Progress

1. If no return privileges exist for C, then the interpretation of *unit-check-in* of C is not valid due to an unsatisfied precondition at level 3. Plan generation can be invoked to satisfy the precondition, thereby making the interpretation valid.

2. If the programmer had issued a command for *unit-check-in* on module B, this action would be recognized as superfluous—its goal is already true (presumably, B was not modified to make S1).

3. If the programmer had attempted to relinquish the return privileges on C (perhaps meaning to do a check-in but garbling the parameters on the command), the action could be "doubly" an error. Return privileges on C are necessary to doing *unit-check-in* of C, which is necessary to completing the *build* of S1. Since the proposed action interferes with other actions, the programmer is asked to confirm it before execution (assuming the interpretation is otherwise valid). However, there may be no valid interpretation for this action (e.g., no "reason" to do a relinquish); then an error would be reported.

   As an example of how planning and plan recognition are complementary, consider the request *achieve built(S1)*. The goal is that of the partially completed *build* plan, so the request equates to completing that plan. If this happens after unit-testing is complete, there will be one remaining subgoal in *build* to satisfy. The *system-check-in* operator (level 2) will be chosen to satisfy it, and *unit-check-in* (level 3) will be chosen to expand two subgoals in *system-check-in*; since B is already checked-in, the third subgoal is vacuously satisfied. Because the programmer does not have return privileges for A, a further expansion of the precondition in *unit-check-in* for A will have to be done. The final plan consists of three actions: two *unit-check-ins*, one of which is preceded by a dummy *check-out* to acquire the return privilege.

   Planning can be invoked on any level goal within an existing plan, or on new goals. Plan recognition sets the context for planning in two ways. First, it provides additional state information not otherwise available (e.g., the effects of *release*, or any effect in a complex operator not logically implied by the conjunction of the subgoals). Second, when the goal is part of a recognized plan in progress, some planning choices, such as parameter bindings, may already be decided.

3 Achieving Deeper Process Understanding

One obstacle to deeper process understanding is that some useful information about the state of the world cannot be directly observed from the actions performed; nor can it be computed with certainty from observable data. Consider the issue of the "releasability" of a given system version. Customers generally expect (bug-free) releases with successively greater functionality. The *release* operator of Figure 2 ignores these considerations—it is too *permissive*. For example, it allows releases

in random order of functionality. Adding a requirement that a new release must have been developed after the current release gives an operator that is too *rigid.* A development time-stamp is not a perfect predictor of functionality, although generally versions developed later have more function. Also, there could be other exceptions, such as re-releasing the previous release when the current release is found to have a serious bug.

Although a "decision procedure" to determine releasability with certainty seems unattainable, it is possible to identify some rules for making plausible assumptions about releasability, based on what is typically the case. These assumptions would then be the basis for determining the *credibility* of a particular *release* action; actions below a certain credibility threshold could be challenged, not as actual errors, but as possible errors. If the programmer confirms the action, the assumptions would need to be revised to be consistent with the programmer's judgment.

Reasoning that involves making and revising plausible assumptions in the absence of complete information is called *non-monotonic reasoning* (NMR) [9]; one form of NMR is a *truth maintenance system* (TMS) [7]. In the next two sections, we show how deeper process understanding can be achieved using a TMS.

3.1 Formalizing Additional Knowledge

A TMS uses a multi-valued logic approach to NMR. It maintains a network of *nodes,* each of which can be labelled IN or OUT. Separate nodes are used for a predicate and its negation. If the node for a predicate is IN and the node for its negation is OUT, the predicate is *true*; if the node is OUT and the negation is IN, the predicate is *false.* If both are OUT, the truth value is *unknown*; if both are IN, there is a *contradiction.*

Justifications capture the relationships between the nodes, correlating a set of *support* nodes and a set of *exception* nodes with a *conclusion* node. A justification of the form A EXCEPT B $\rightarrow$ C means if A is IN and B is OUT, then C is IN. The exception node B represents the non-monotonic content of the justification; a monotonic justification (standard logical implication) has an empty list of exceptions. In order for a node to be IN, it must have at least one *valid* justification; a justification is valid if all its support nodes are IN and all its exception nodes are OUT. A *premise* justification has empty support and exception lists; it is always valid.

Process knowledge about releasability is given in justification form in Figure 7 (left column). Rule J1 covers the common situation: a non-buggy version developed after the current release is considered releasable. Rule J2 covers a re-release scenario; if the current release is buggy, the previous release is considered releasable unless it is buggy. Rule J3 provides that if rules J1 and J2 don't apply, a version is considered

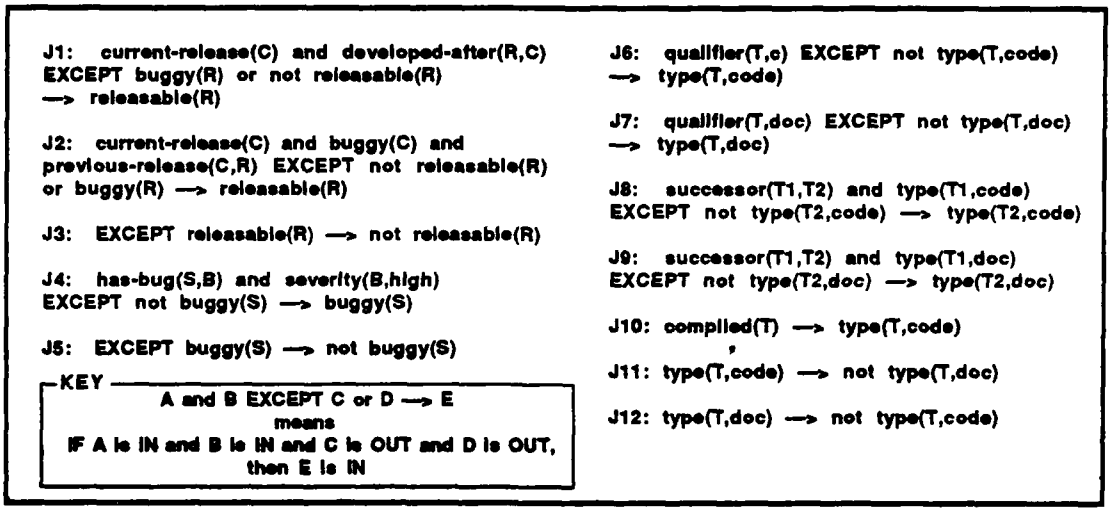


Figure 7: TMS Justifications

not releasable.[1] Rules J4 and J5 give a (simplistic) definition of the predicate buggy, based on the descriptions of explicitly reported bugs in a bug database. An additional set of justifications (Figure 7, right column) shows how the *type* of a text file can be deduced (based on either the qualifier on its file name or the type of its predecessor). This set contains monotonic justifications; for example, if the file has compiled successfully, then it is definitely code.

These justifications provide a way to compute the truth values of new predicates (*releasable, buggy, and type*), thus enlarging the state of the world beyond the original, *core* state. When the justifications are instantiated, and the truth values of core predicates entered (with premise justifications), the truth maintenance process will label the nodes, giving a read-out on the truth values of the new predicates. When the core state changes as a result of an action, the nodes will be relabelled and the new predicates may change. The TMS will also determine whether a node is *certain* or *by-assumption*. Nodes belonging to the core state are always certain. Other nodes are certain if they are the conclusions of monotonic justifications, all of whose support nodes are certain. Remaining nodes are by-assumption; for nodes that are IN by-assumption, one or more non-monotonic rules were needed to justify them. (The labels of nodes that are certain cannot be changed when assumptions are being revised.)

A TMS labelling is shown in Figure 8, where Sys2 is the current release, Sys1 the

[1] A justification of the form EXCEPT A $\rightarrow \neg$ A is only made valid *after* it has been determined that there are no valid justifications for A. This is a special case of a more general feature that allows justifications to be prioritised according to their predictive accuracy. Priorities ensure that stronger justifications are not invalidated by weaker ones, as described in [13].

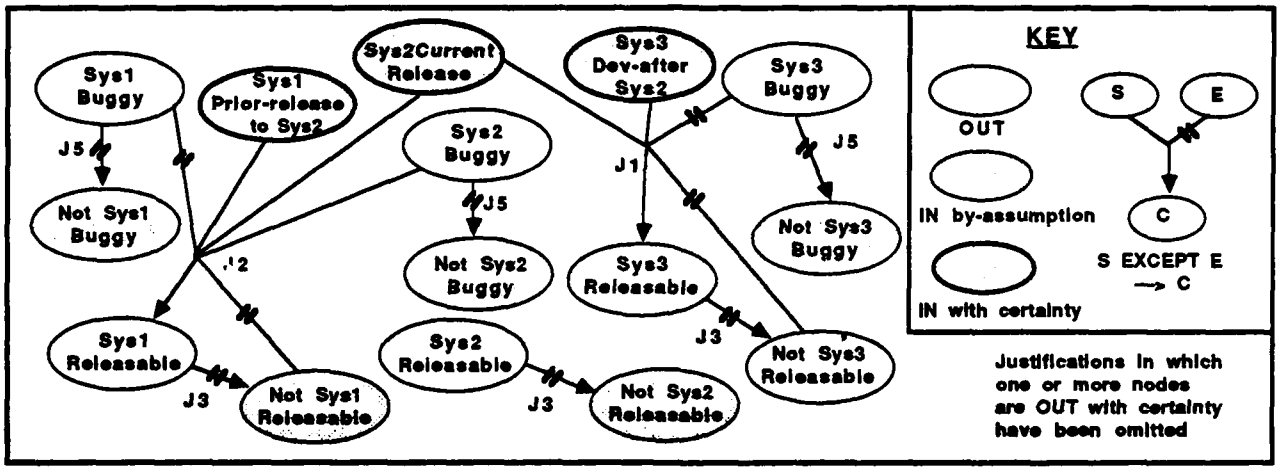


Figure 8: A TMS Labelling

previous release, Sys2 was developed after Sys1, and Sys3 was developed after Sys2. In the absence of any reported, high-severity bugs, none of these systems is buggy (J5 valid, J4 not valid in each case). Sys1 is not releasable—neither J1 nor J2 are valid, but J3 is valid. Sys3 is releasable by J1. Sys2 was at one time releasable, but this is no longer the case.

Since the justifications determine the truth status of the new predicates, *releasable* can be used wherever needed in operator definitions. In particular, *releasable(System)* can now be added as a constraint in the *release* operator of Figure 2.

3.2 Credibility and Revision of Assumptions

In the enlarged world state that is now accessible, predicates will evaluate to one of five truth values: true with-certainty, true by-assumption, unknown, false by-assumption, or false with-certainty. These correspond to five credibility classes, informally described as certainly OK, credible, can't tell, not credible, and certainly not OK. As the preconditions and constraints of an action are checked along the path from the action to a top-level goal, the credibility rating can be accumulated.

Credibility can be used in two ways: to select among competing interpretations for an action and to flag potential errors in actions. For example, if a file is assumed to contain code, we expect to see it used in actions on code, not actions on documentation. When there is an action involving this file that could be either part of preparing code or part of preparing documentation, the interpretation for preparing code will have a higher credibility, and will be preferred. Potential errors flagged

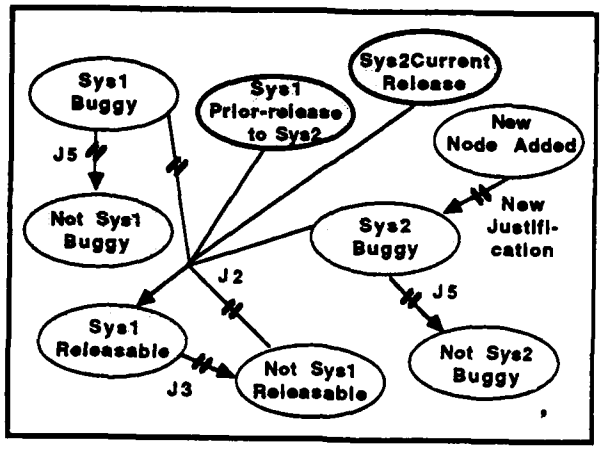


Figure 9: Revised Assumptions

for the programmer's attention correspond to interpretations with a "not credible" rating.

The internal structure of a TMS is designed for revising assumptions, not just reporting assumptions. When proceeding with an interpretation in which one or more predicates evaluated to unknown or false by-assumption, it is necessary to make the world state consistent with the requirements of the action. This can be trivially accomplished by adopting the desired assumptions; but then, clues that other assumptions are wrong will be ignored. Since the justifications provide the accepted rationale for various assumptions, it is better to find a rationale for the desired assumption than to adopt it outright. This will "integrate" the programmer's judgment into the current set of assumptions.

Consider an action (re-)releasing Sys1 in the state diagrammed in Figure 8. Since *releasable(Sys1)* is false by-assumption, this action will be challenged. If the programmer confirms this action, then *releasable(Sys1)* must be made true. Only two rules, J1 and J2, justify *releasable.* J1 cannot be made valid—it has a support node (*developed-after(sys1,sys2)*) that is OUT with certainty. J2 can be made valid by making *buggy(Sys2)* IN. This has to be done by supporting *buggy(Sys2)* directly, since no revision of assumptions can make J4 valid. This change and the new labels are shown in Figure 9. The algorithm for revising assumptions [13] is similar to dependency-directed backtracking [7].

This example shows how the process assistant uses the justifications to make independent assessments (originally to conclude that Sys1 is not releasable), and yet accede to a different decision and supply a rationale for that decision (that Sys1 is releasable *because Sys2 must be buggy).*

4 Status and Future Work

We have built a testbed for experimentation with the GRAPPLE plan-based process assistant. Two versions of the plan recognizer have been implemented. The TMS and related facilities for deeper process understanding are implemented in a Prolog version of the plan recognizer. GRAPPLE is not tied to a particular software environment; rather, it accepts command streams transcribed from actual terminal sessions or fabricated for experimental purposes. We have studied actual session transcripts to develop a better understanding of the content of process definitions, but have not yet tested GRAPPLE in a real setting.

There are several dimensions along which plan-based process support can be extended. We have already developed (and are currently implementing) a method of using metaplans to represent error recovery strategies and other types of knowledge of exceptional situations [12]; this is particularly important in light of the "trial and error" character of software development. We also want to explore the feasibility of applying intelligent assistance to the commands within an interactive tool (e.g., a syntax-directed editor). This would extend support to lower-level process activities and capitalize on synergy between levels, as the external context affects what is done within the tool, and vice versa. The most exciting extension involves the use of multi-agent planning techniques to represent project-level coordination, cooperation, and negotiation among programmers.

5 References

1. Balzer, R.M. "Living in the Next Generation of Operating System." *IEEE Software*, 4:6 (November 1987), 77-85.
2. Balzer, R.M.; Cheatham, T.E.; and Green, C. "Software Technology in the 1990's: Using a New Paradigm." *IEEE Computer* (November 1983), 39-45.
3. Bernstein, P.A. "Database System Support for Software Engineering." *Ninth International Conference on Software Engineering* (March 1987), 166-178.
4. Bisiani, R.; Lecouat, F.; and Ambriola, V. "A Planner for the Automation of Programming Environment Tasks." *Twenty-first International Hawaii Conference on System Sciences* (January 1988).
5. Chen, P.P. "The Entity-relationship Model: Toward A Unified View of Data." *ACM Transactions on Database Systems*, 1:1 (March 1976), 9-36.
6. Croft, W.B. and Lefkowitz, L.S. "Knowledge-based Support of Cooperative Work." *Twenty-first International Hawaii Conference on System Sciences*, (January 1988) 312-318.

7. Doyle, J. "A Truth Maintenance System." *Artificial Intelligence*, 12 (1980), 231-272.

8. Fickas, S.F. "Automating the Transformational Development of Software." *IEEE Transactions on Software Engineering*, SE-11:11 (November 1985), 1268-1277.

9. Genesereth, M.R. and Nilsson, N.J. *Logical Foundations of Artificial Intelligence*. Palo Alto, California: Morgan Kaufmann, 1987.

10. Huff, K.E. "A Database Model for Effective Configuration Management." *Proceedings of Fifth International Conference on Software Engineering* (March 1981), 54-61.

11. Huff, K.E. and Lesser, V.R. "Knowledge-based Command Understanding," Technical Report 82-6, Department of Computer and Information Science, University of Massachusetts, Amherst, 1982.

12. Huff, K.E. and Lesser, V.R. "Metaplans That Dynamically Transform Plans," Technical Report 87-10, Department of Computer and Information Science, University of Massachusetts, Amherst, 1987.

13. Huff, K.E. and Lesser, V.R. "Plan Recognition in Open Worlds," Technical Report 88-18, Department of Computer and Information Science, University of Massachusetts, Amherst, 1988.

14. Johnson, W. and Soloway, E. "PROUST: Knowledge-Based Program Understanding." *IEEE Transactions on Software Engineering*, 11:3 (March 1985), 267-275.

15. Kaiser, G.E. and Feiler, P.H. "An Architecture for Intelligent Assistance in Software Development," *Proceedings of the Ninth International Conference on Software Engineering* (1987), 180-188.

16. Leblang, D.B. and Chase, R.P. "Computer-aided Software Engineering in a Distributed Workstation Environment." *Proceedings of SIGSOFT/ SIGPLAN Symposium on Practical Development Environments* (1984), 104-112.

17. Luria, M. "Goal Conflict Concerns." *Proceedings of IJCAI* (August 1987).

18. Rich, C. and Shrobe, H.E. "Initial Report on a Lisp Programmer's Apprentice." *IEEE Transactions on Software Engineering* SE-4 (November 1978).

19. Osterweil, L. "Software Processes are Software Too." *Proceedings of the Ninth International Conference on Software Engineering* (1987), 2-13.

20. Penedo, M.H. and Stuckle, E.D. "PMDB—A Project Master Database for Software Engineering Environments." *Proceedings of the Eighth International Conference on Software Engineering* (1985), 150-157.

21. Ramamoorthy, C.V.; Usuda,Y.; Tsai, W.T.; and Prakash, A. "GENESIS: An Integrated Environment for Supporting Development and Evolution of Software," *IEEE Ninth International Computer Software and Applications Conference* (October 1985), 472-479.

22. Sacerdoti, E.D. *A Structure for Plans and Behavior.* New York: Elsevier-North Holland, 1977.

23. Tate, A. "Project Planning Using a Hierarchical Non-linear Planner." Department of Artificial Intelligence Report 25, Edinburgh University, 1976.

24. Tichy, W.F. "RCS—A System for Version Control." *Software Practice and Experience*, 15:7 (July 1985), 637-654.

25. Waters, R.C. "The Programmer's Apprentice: A Session with KBEmacs." *IEEE Transactions on Software Engineering*, SE-11:11 (November 1985), 1296-1320.

26. Wile, D.S. and Allard, D.G. "Worlds: An Organizing Structure for Object-Bases." *Proceedings of Second SIGSOFT/SIGPLAN Symposium on Practical Development Environments* (1986), 16-26.

27. Wilensky, R.; Arens, Y.; and Chin, D. "Talking to UNIX in English: An Overview of UC." *CACM*, 27:6 (June 1984), 574-593.

28. Wilkins, D.E. "Domain-Independent Planning: Representation and Plan Generation." *Artificial Intelligence*, 22 (1984), 269-301.

29. Williams, L. "Software Process Modeling: A Behavioral Approach." *Proceedings of the Tenth International Conference on Software Engineering* (1988), 174-186.

Appendix 6-B

Planning for the Control of an Interpretation System

Norman Carver and Victor Lesser

Abstract: Interpretation is a complex and uncertain process which requires sophisticated evidential reasoning and control schemes. We have developed a framework which models interpretation as a process of gathering evidence to manage uncertainty. The key components of the approach are a specialized evidential representation system and a control planner with heuristic focusing. The evidential representation scheme includes explicit, symbolic encodings of the sources of uncertainty in the evidence for the hypotheses. This knowledge is used by the control planner to identify and develop strategies for resolving the uncertainty in the interpretations. Since multiple, alternative strategies may be able to satisfy goals, the control process can be seen to involve a search. Heuristic focusing is applied in parallel with the planning process in order to select the strategies to pursue and control the search. The control plan framework allows the use of a flexible focusing scheme which can switch back and forth between strategies depending on the nature of the developing plans and changes in the domain.

This work was supported by the Air Force Systems Command, Rome Air Development Center, Griffiss Air Force Base, New York 13441-5700, and the Air Force Office of Scientific Research, Bolling AFB, DC 20332 under Contract No. F30602-85-C-0008. This contract supports the Northeast Artificial Intelligence Consortium (NAIC).

1 Introduction

Interpretation is the process of determining a high-level, abstract view of a set of data based on a hierarchical specification of possible viewpoints. Hypotheses representing interpretations of some subset of the data are developed using the hierarchical support relations to identify the legal "evidence" for the hypotheses. Complex interpretation tasks require sophisticated evidential reasoning and control schemes which can deal with the uncertainty inherent in the interpretation process. For example, combinatorial considerations preclude complete construction and evaluation of all potential interpretations: control must be exercised over the creation and refinement of hypotheses. This means that the system cannot be sure whether it has even created all the correct interpretations and must compare alternative hypotheses based on partial knowledge. In many domains the uncertainty is compounded by the volume of data being too large to be completely considered and/or by the data being uncertain and possibly incorrect.

We have developed an interpretation framework based on a model of interpretation as a process of gathering evidence to manage uncertainty. The key components of the approach are a specialized evidential representation system and a control planner with heuristic focusing. The evidential representation scheme includes explicit, symbolic encodings of the *sources of uncertainty* in the evidence for the hypotheses. That is, evidence provides uncertain support for a hypothesis because there are conditions under which the evidence may fail to support the hypothesis: the sources of uncertainty in the evidence. For example, while some piece of sensor data in a vehicle monitoring system can be used to support a particular vehicle hypothesis, the resulting evidence is uncertain because the data may actually be due to a sensor malfunction or may support a competing, alternative vehicle hypothesis. Our model of interpretation associates a set of sources of uncertainty with the evidence for the hypotheses: partial evidence, uncertain evidence inference, uncertain evidence premise, alternative evidence interpretation (which represents the relations between alternative hypotheses), conflicting evidence, etc. This knowledge is used by the control process in elucidating strategies for meeting the problem-solving goals since the *purpose* of interpretation actions is to resolve the uncertainty in the hypotheses. The evidential representation scheme is discussed in detail in section 2.1.

Control decisions are made through an incremental planning process which identifies, selects, and implements problem-solving strategies. The available strategies are defined as a hierarchy of *control plans*. Control planning involves determining the subgoals of the current plan which must be satisfied next and then matching each subgoal to the possible control plans to determine how it might be satisfied. Primitive control plans represent actions and have corresponding functions for executing the actions. Planning and execution are interleaved—that is, the plans are only elaborated to the point of selecting the next action—because the outcome of actions is uncertain (e.g., the attempt to infer support for a hypothesis from some data may fail). In general, there will be many partial control plan instances which could be further elaborated at any point in the processing—i.e., many possible strategies for pursuing the system goals. The alternative

control plan instances represent the choices of which hypotheses to resolve uncertainty in, what sources of uncertainty to eliminate, and how to do it. Thus, one of the major issues for interpretation systems is the development of an effective focus-of-attention scheme. In our system focusing is accomplished as part of the multi-stage process of refining and elaborating the control plans. Maintaining a framework of control plan instances gives the system a great deal of control flexibility. The focusing process can move back and forth between strategies by refocusing in the plan instance hierarchy based on the characteristics of the developing plans and factors such as data availability. The details of control planning and focusing are discussed in section 2.2.

The work presented here grew out of our experience developing a focusing scheme for plan recognition [1]. Plan recognition systems have tended to ignore the practical aspects of focusing the interpretation process and of comparing alternative hypotheses to determine just what it is that the system believes. The scheme we developed used an explicit record of the application of focusing heuristics to guide the system and allow it to revise its interpretations. However, the granularity of the control was too coarse for many domains, the assumption information was of limited value in controlling backtracking, and hypothesis uncertainty was confused with the control decisions (see [2]). Of interest to us also has been work on planning for control by Clancey [3], Hayes-Roth [6], and Durfee and Lesser [5]. However, none of these systems provide a completely suitable framework for interpretation. Clancey's tasks and meta-rules are really control plans and their substeps. The framework is limited by the fact that meta-rules directly invoke subtasks so there is no ability to search for and consider alternative strategies; focusing is implicit in the meta-rule preconditions. Of course, this may be fine for diagnosis problems which tend to be more exhaustive than interpretation systems. The Hayes-Roth work on blackboard systems for control is more general than our work (which concentrates on interpretation). However, this generality means that little guidance is provided in how to structure control knowledge— i.e., the need for uncertainty knowledge as a basis for elaborating control plans. Another drawback of the control blackboard approach is its reliance on an agenda mechanism which must consider all possible actions on each loop. By contrast, focusing in parallel with the control plan hierarchy as we do provides, in effect, a partitioned agenda: only those actions immediately applicable to the in-focus plan or goal are considered. The incremental planning approach of Durfee and Lesser builds abstract models of the interpretation data and uses these models to guide further processing. This idea can be handled as one type of problem-solving strategy in our system with the addition of appropriate abstract operators and control plans representing the strategy. Most work on evidential representation systems relies on having a fixed set of alternatives among which to partition belief (as in diagnosis problems). This is not the case for interpretation (see section 2.1) so such work is not directly applicable. Our use of symbolic representations of uncertainty was inspired, in part, by Cohen's work on symbolic representations of evidence called *endorsements* [4].

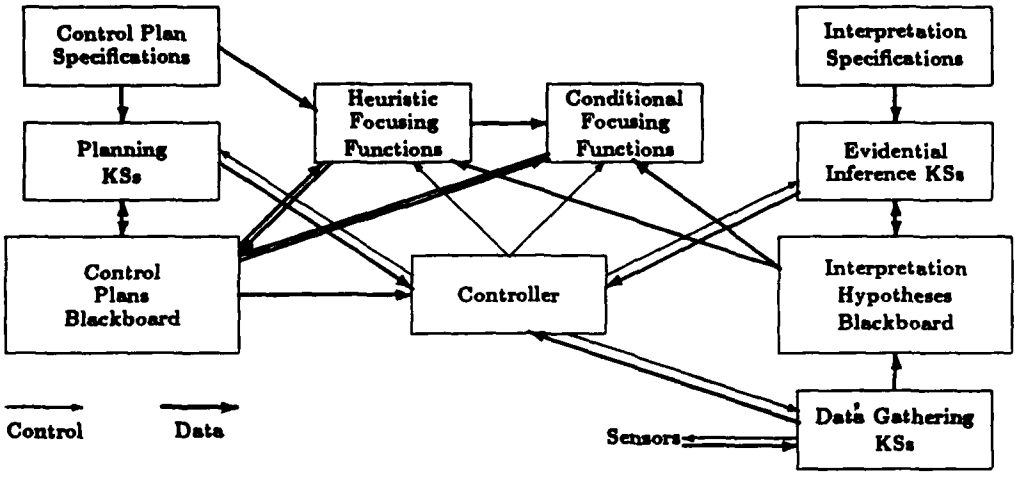


Figure 1: Architecture of System

2 Planning to Resolve Uncertainty

Figure 1 illustrates the major components of our system. The controller is responsible for executing the basic control loop (see Figure 4). Control plan instances created by the planning process are maintained on the Control Plans Blackboard. Expansion of the control plans is accomplished using Planning KSs based on the control plan specifications. Interpretation hypotheses are developed by executing interpretation actions using the corresponding Evidential Inference KSs and the Data Gathering KSs. The hypotheses are maintained on the Interpretation Hypotheses Blackboard along with information about the evidence supporting the hypotheses, the uncertainty in that evidence, and the relations between the hypotheses.

2.1 Evidence and Sources of Uncertainty

For each interpretation hypothesis, H, the hierarchy specification defines both a set of sets of lower-level, support hypotheses, $\{\{S_i\}_j\}$, and a set of higher-level, explanation hypotheses, $\{E_i\}$. Hypotheses may be supported by multiple sets of supporting hypotheses to model those domains in which there can be multiple *sources of evidence*—for example, in vehicle monitoring where vehicles might be supported by data from a variety of sensor types. The lowest-level units in the hierarchy correspond to the data to be interpreted, while the highest-level units represent the most abstract interpretation of that data (these hypotheses have the explanation of "top-level"). Hypotheses also include parameters whose values are constrained and defined by its support and explanation hypotheses.

The model of evidence that we use is based on the requirements for control—i.e., that the *sources of uncertainty* for each hypothesis can be used to drive the control process. Each hypothesis is based on a *set* of evidential inferences of the form: $S_{k_l} \Rightarrow H$ (supporting

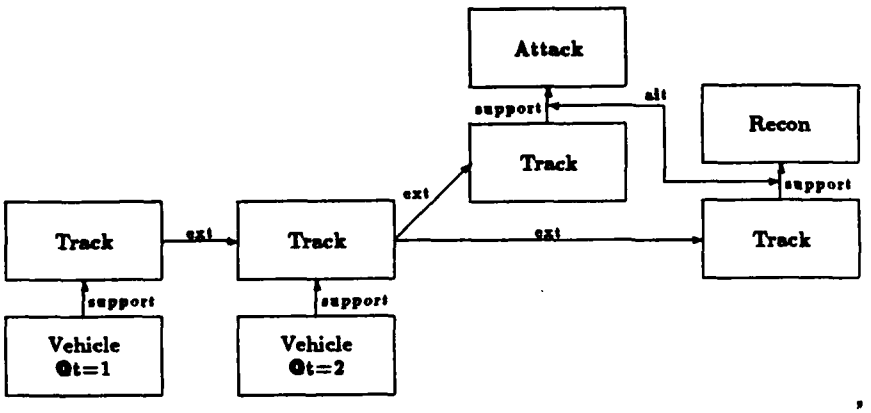


Figure 2: Hypothesis Representations

evidence) or $E \Rightarrow H$ (explanation evidence) where $S_{k_l} \in \{S_i\}_l$, $\{S_i\}_l \in \{\{S_i\}_j\}$, and $E \in \{E_i\}$. There are then the following potential classes of sources of uncertainty in each hypothesis:

- The supporting evidence may be incomplete—i.e., $\{S_j\}_l \subset \{S_i\}_l$.
- There may be no explanation evidence.
- The premise hypothesis of an evidential inference may be uncertain—i.e., some S_{k_l} or E_k is uncertain (based on uncertain evidence).
- There may be alternative interpretations for the evidence–i.e., for some $S_{k_l} in \{S_i\}_l$ the correct inference is $S_{k_l} \Rightarrow H'$.
- It may be uncertain whether an inference satisfies the hierarchy constraints—that is, it is uncertain whether $\{S_j\}_l \subset \{S_i\}_l$.
- Conflicting evidence–i.e., some S_{k_l} doesn't exist.

Specific instances of these classes of uncertainty are instantiated as symbolic expressions and attached to the hypotheses. For example, the Attack hypothesis in figure 2 includes the expression (Alt-Interp "track" "recon") to represent the uncertainty in its supporting track evidence due to the existence of an alternative interpretation for it, the Recon hypothesis. We have also used this framework to extend the knowledge typically found in an interpretation system by including sources of uncertainty for evidential relations which do not result from alternative interpretations. For example, acoustic sensor data in a vehicle monitoring system causes uncertainty because the data may result from a sensor malfunction or from weather disturbances. We represent these factors as sources of uncertainty even though these factors are not used as interpretation hypotheses.

One of the key characteristics of interpretation problems which distinguishes them from diagnosis problems is that evidence not only *justifies* the interpretation hypotheses, it also *refines* them by constraining the parameters of the hypotheses. That is, we don't simply gather evidence to decide among a set of predetermined alternatives; we must gather evidence just to determine what the set of alternatives consists of. For example, in vehicle monitoring, sensor evidence for vehicle hypotheses not only supports the belief

Name	Eliminate-Sources-of-Uncertainty
Description	Eliminates the sources of uncertainty from the hypothesis ?hyp until the belief in ?hyp is greater than ?belief.
Goal Form	(Have-Eliminated-SOUs ?hyp ?belief)
In Variables	(?hyp ?belief)
Out Variables	nil
Temp Variables	?sou
Subgoals	(Have-Source-of-Uncertainty Have-Eliminated-SOU)
Definitions	((Have-Source-of-Uncertainty . (Have-SOU ?hyp ?sou)) (Have-Eliminated-SOU . (Have-Eliminated ?hyp ?sou)))
Sequence	(ITERATION (GREATER (belief ?hyp) ?belief) (SEQUENCE Have-Source-of-Uncertainty Have-Eliminated-SOU))
Constraints	nil

Figure 3: Control Plan Specification

in a vehicle, it also constrains the vehicle type and position. As a consequence, multiple versions of each hypothesis, called extensions, must be used to represent the various hypothesis refinements supported by the (uncertain) evidence. As part of our representation of the uncertainty in the hypotheses, we have developed a scheme for representing the alternative extensions of hypotheses and the interrelations between these extensions. Figure 2 shows simple example of such an extension framework. In this example, there are four extensions of the Track hypothesis, each of which is caused by the addition of evidence which constrains the Track. Of particular interest are the alternative extensions created by the competing interpretations of Track as support for an Attack mission or a Recon mission. Notice that these alternative extensions are used to recognize the relations between hypotheses. The advantage of this framework is that we can simultaneously reason about all the possible extensions of a hypothesis and the uncertainty which results from these competing extensions.

2.2 Control Plans and Heuristic Focusing

Control plans are defined using specifications like the one in Figure 3. The goal of the plan is specified by the Goal Form. Variables in the goal form may be either Input Variables which must be supplied to the plan or Output Variables which are bound upon completion of the plan. Each plan is realized by a sequence of subgoals which is specified in the Sequence clause. This clause uses a shuffle grammar to express strict sequences, concurrency, alternatives, optional subsequences, and iterated subsequences. The sequencing constraints of the Sequence clause can be augmented with additional ordering constraints and constraints on the subgoal variable values in the Constraints clause. The subgoal Definitions clause specifies the goal form for each subgoal and whether the variables are input or output variables. Subgoal output variables allow subgoals to return the results of executing actions. Many actions are capable of returning multiple bindings for variables, for example, when determining an available sensor unit. To deal with this fact, variables are allowed to take on *multiple valued* values which represent a set or range of bindings. Focusing then determines the value(s) to use in subsequent plan expansion.

repeat: Pursue-Focus on each element of Current-Focus-Set until nil.

Pursue-Focus(focus)
case on type(focus):

plan	Focus on variable bindings to select plan instances for Current-Focus-Set.
	Expand plan instances to next subgoals.
	Focus on subgoals to select subgoals for Current-Focus-Set.
subgoal	Match goal to applicable plans.
	Focus on plans to select new focus elements for Current-Focus-Set.
primitive	Execute primitive plan action to get result.
	Update plan states and select new focus element for Current-Focus-Set.
	Check refocus and subgoal conditions.

Figure 4: Control Planning Loop

The basic control planning loop is detailed in Figure 4. An example control plan instance is represented in Figure 5 as an AND/OR tree of plan and subgoal nodes. The situation represented is such that the top-level plan to solve the interpretation problem has created a subgoal of resolving the uncertainty in the Attack hypothesis of Figure 2. In order to pursue this subgoal, the subgoal form is unified with the goal forms of the defined control plans to determine which plans are applicable to satisfying the subgoal. In this case, only a single plan is relevant to satisfying this goal, Eliminate-Sources-of-Uncertainty, the plan whose specification was shown in Figure 3. In general, there would be multiple matches corresponding to multiple possible strategies for satisfying the goal and focusing knowledge would be applied to select the plan(s) to focus on. Pursuing an in-focus control plan means expanding the control plan to create subgoals representing the substeps of the plan which need to be satisfied next. The first two subgoals of the Eliminate-Sources-of-Uncertainty plan are shown in Figure 5 even though they are sequential steps (signified by the horizontal arrow between their arcs). The first subgoal, Have-SOU, can be satisfied by the primitive plan, Get-Sources-of-Uncertainty. Primitive plans represent actions which may be carried out with corresponding Knowledge Sources. Primitives may generate information for the planning process (e.g., determining an available sensor to generate new data) or may generate interpretation evidence (e.g., to create an evidential inference). Actions may fail or may succeed and return results. In the case of Get-Sources-of-Uncertainty here, the action succeeds and returns a multiple-valued value consisting of the symbolic representations of the sources of uncertainty in the attack hypothesis. This result is bound to the variable ?sou of the primitive and the status of the plan is set to complete. The outcome of the action i.e., the change in status and the variable bindings is then propagated to the subgoal the primitive satisfies and then in turn to the plan containing the subgoal. The change in the state of the subgoal may mean that this plan has failed or succeeded which would cause additional propagation. In this case, it simply changes the state of the plan so that it is expecting the next subgoal and binds the variable ?sou. Multiple-valued values such as that just bound to ?sou are used to represent a set of alternative bindings for a variable–i.e., uncertainty over the correct value. Because ?sou is used as an input for the next subgoal, Eliminate-SOU, it is necessary to apply focusing knowledge at this point to select the value(s) to be used

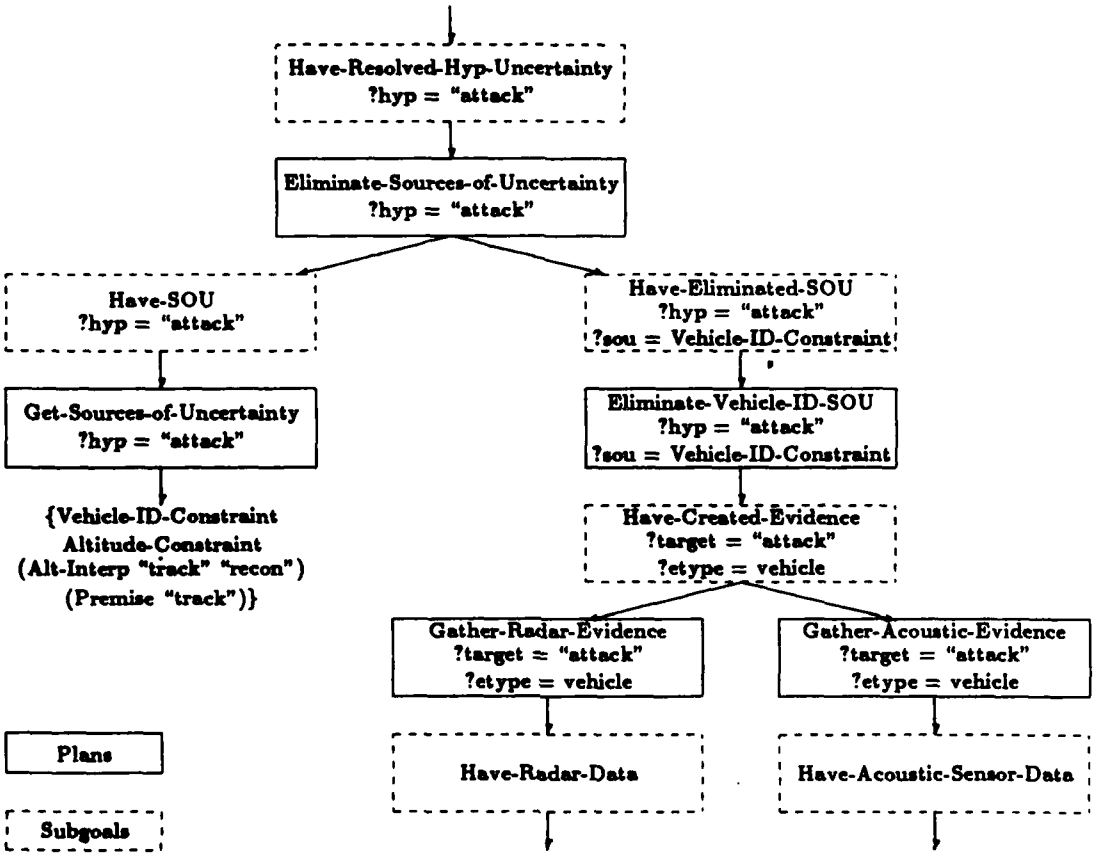


Figure 5: Control Plan Instance

for further expansion. The single-value version(s) of the subgoal are then used to select applicable control plans. In the example, the heuristic focusing knowledge associated with the variable ?sou has selected the source of uncertainty Vehicle-ID-Constraint as the in-focus binding for ?sou.

Focusing heuristics represent meta-level knowledge relative to the knowledge in the control plans. Whereas control plans embody problem-solving strategies for interpretation, focusing heuristics embody strategies for selecting the appropriate problem-solving strategies. In our framework, focusing heuristics are associated with particular control plans so they are specific to the context of the plans. Since there are several points at which focusing decisions are made, we provide three different types of focusing knowledge: subgoal, variable, and updating. Subgoal focusing knowledge is associated with the plan in general and is used to select among competing subgoals. Variable focusing knowledge is associated with each of the variables of the control plan environment and is used to select among competing bindings. Updating focusing knowledge is associated with each type of control plan subgoal and used to decide how to proceed when a plan relevant to the subgoal completes–whether it succeeds or fails. This knowledge controls backtracking by deciding whether to accept the result and propagate it or pursue existing

alternatives. The heuristics are represented as procedural code, but this is the subject of current research (see section 3).

The basic control process described above is highly top-down, depth-first. However, the uncertainty of interpretation requires a strong bottom-up component as well. We accomplish this with several extensions to the basic focusing scenario. These extensions make it possible for the system to shift its focus between competing strategies in response to the characteristics of the developing plans and factors such as data availability. Focusing is extended by allowing decisions to be: absolute, postponed, or preliminary. Absolute focusing heuristics simply select a single path to be pursued–subject, of course, to plan failure (which is handled by the updating process). However, focusing heuristics may not always be able to select a single "best" path to pursue. Instead, they may need to partially expand each of several competing strategies to gather more specific information about the situation. We handle this by allowing multiple paths to be expanded with postponed focusing. A multiple-focus form is created which specifies the paths to be pursued, the conditions for refocusing, and the refocus handler. Refocus conditions are evaluated following the execution of any action (only actions generate information). When they are satisfied, the refocus handler is invoked and re-evaluates the choices in the new context to eliminate the multiple foci. An example of a postponed focusing decision occurs later in the refinement of the control plan in Figure 5. There are two control plans applicable to satisfying the subgoal Have-Created-Evidence: Gather-Radar-Evidence, which uses radar data to create the desired evidence, and Gather-Acoustic-Evidence, which uses acoustic sensor data. The system is uncertain about how to proceed because it cannot be sure which source of evidence will provide the best track evidence without knowing more about the actual data which is available. To handle this situation, the focus decision is postponed until the first subgoal of each alternative plan is satisfied–that is, until the potential data is determined. The refocus handler is then used to evaluate the focusing alternatives in light of the additional information–(e.g., by evaluating the relative quality of the available data for each alternative plan).

Preliminary focus decisions are similar to postponed decisions except that only a single alternative is pursued. If the refocus conditions are satisfied, the original focus options are re-evaluated by the focus handler. Preliminary focus decisions are used when one alternative is likely to be the best, subject to certain reservations about its progress. They can also be used to limit the amount of effort expended on one alternative by including refocus conditions which set a limit on the amount of time to be expended or the level of completion to be reached.

3 Conclusion and Status

This work is a further example of the utility of making control decisions through planning. It expands on existing research with respect to planning for the control of an interpretation system in three significant ways: the control task is viewed as being driven by

the need to resolve uncertainty, the uncertainty of the interpretation hypotheses is represented explicitly and symbolically, and the process of finding the correct control plan may be seen to involve a search process which requires focusing. The combination of a control plan with parallel focusing and a symbolic representation of the interpretation uncertainty provides a flexible framework which can be used to implement sophisticated control strategies.

We presently have a prototype implementation of this framework which simulates a system for monitoring aircraft. A variety of data sources such as acoustic sensors, radar, and emissions detectors are included. Additional sources of evidence such as terrain, air defense positions, and weather information are also available. Active control over evidence gathering can be effected through control of the operations of some of the sensors. Interpretation hypotheses cover a variety of missions including those involving coordination of multiple aircraft.

One of the areas of current research in this project is the issue of languages for expressing focusing knowledge. Up to now we have viewed focusing knowledge as expert-level knowledge specific to the control plans and have simply expressed it procedurally. However, the need for a language with an appropriate set of primitives is clear. In particular, we are looking into the factors needed for focusing in real-time applications. This includes such things as estimated processing and elapsed times and estimates of the quality of the evidence from alternative strategies.

References

[1] Carver, Norman, Victor Lesser, and Daniel McCue, "Focusing in Plan Recognition," *Proceedings of AAAI-84*, 1984, 42–48.

[2] Carver, Norman, *Evidence-Based Plan Recognition*, TR 88-13, Computer and Information Science Department, University of Massachusetts, 1988.

[3] Clancey, William, "From GUIDON to NEOMYCIN and HERACLES in Twenty Short Lessons," *AI Magazine*, 7 (3) 1986, 40–60.

[4] Cohen, Paul, *Heuristic Reasoning About Uncertainty: An Artificial Intelligence Approach*, Pitman, 1985.

[5] Durfee, Edmund, and Victor Lesser, "Incremental Planning to Control a Time-Constrained, Blackboard-Based Problem Solver," *IEEE Transactions on Aerospace and Electronic Systems*, September, 1988.

[6] Hayes-Roth, Barbara, "A Blackboard Architecture for Control," *Artificial Intelligence*, 26, 1985, 251–321.

Appendix 6-C

Planning and Execution of Tasks in Cooperative Work Environments

Lawrence S. Lefkowitz and W. Bruce Croft

Abstract: Problem solving has long been recognized as an important component in many work environments. The potentially complex and often cooperative nature of even apparently "routine" tasks has led to attempts to use planning techniques to support work in real-world domains. This paper describes the work being done by the POLYMER project to construct a planning system to assist in the performance of multiagent, loosely-structured, underspecified tasks. Specifically, we present a representation for modeling tasks, agents and objects within such environments and describe the architecture and implementation of a planning system which uses these models to support cooperative work. We conclude with a description of how this planning system is being used to support further research in areas such as exception handling, negotiation and knowledge acquisition.

This research was supported in part by a contract with Ing. C. Olivetti and by the Air Force Systems Command, Rome Air Development Center, Griffiss Air Force Base, NY 13441-5700.

1 Introduction

Problem solving has long been recognized as an important component in many work environments[1,11,12]. The potentially complex nature of even apparently "routine" tasks has motivated the use of planning techniques to support work in real-world domains[5,17]. However, the work in many environments is cooperative in nature. In such settings, tasks often cannot be performed by an individual; the coordinated effort of a group of people is needed to accomplish a desired goal. The size and complexity of certain tasks and the limited abilities, knowledge, skills and resources of any individual often make a cooperative approach the only way to achieve results. Offices, design teams, management structures, and factories are all examples of cooperative work environments.

As the scope and complexity of the problems addressed by computer-based support systems grows, the need for planning and knowledge-based approaches becomes more apparent. While traditional tools have been adequate to support single-person, small scale tasks (e.g., mail systems and forms tools in offices, compilers and debuggers in software development, etc.), supporting cooperative work requires the coordination of multiple agents using a variety of tools. By using a model of the tasks, objects and agents in an application domain to generate multiagent plans, the POLYMER planning system [6,7,10] can coordinate interdependent activities in complex, underspecified domains.

Cooperative tasks are often "loosely structured" in that there may be a *typical* way (or ways) in which certain goals are accomplished, but the specifications are far from algorithmic. For instance, some steps within a task may be optional, there may be only a partial ordering of steps, any of several tasks may be used to achieve a goal, various agents may be able to perform a particular task, etc. The domain representation language and the associated planner must be able to capture and utilize any structure that is available but must also be able to cope with this flexibility.

This approach to supporting cooperative work addresses important research issues arising in several related fields, including distributed AI, the coordination of multiple agents (both cooperating and competing), and the attempt to reconcile strategic planning with situated actions[13,14,16,18].

The use of planning to support cooperative work has just begun to be explored. The POLYMER project has focused on constructing a planning system to assist in the performance of multiagent, loosely-structured, underspecified tasks. Such a system requires flexible models of the activities and objects in the application domain, and a means of using these models to help users achieve their goals.

In addition to serving as an aid to performing cooperative tasks, the POLYMER planner is being used as a testbed to support further research in the development of cooperative work environments (e.g., the SPANDEX, DACRON and GENEVA systems described in Section 4), and the development of advanced application domains (e.g., office automation environments, being developed both in our own research environment and by independent researchers).

In this paper we present an overview of the POLYMER system, show its use in

supporting cooperative work, and describe the current research centered around POLYMER. The following section describes the architecture and functionality of the POLYMER system. It presents POLYMER's model of an application in terms of an example from the journal editing domain and shows how this model is used by the planner. Section 3 describes the planning process. It shows how a plan is interactively generated and what happens when problems arise. Section 4 presents a brief overview of research projects which are extending the POLYMER planner to develop an integrated cooperative work environment. Finally, the current status of the POLYMER project is summarized in Section 5.

2 The POLYMER system

Over the past two years, the POLYMER system has been developed as a testbed for supporting cooperative work. Using descriptions of domain tasks and objects, POLYMER combines strategic and reactive planning [18] and interacts with its users to generate a plan to accomplish a specified goal. POLYMER's domain description language evolved from one developed for the POISE intelligent interface system [9]. Using this formalism, POLYMER performs the type of hierarchical, non-linear planning described in NONLIN [19] and SIPE [20].

The POLYMER system has been developed using the KEE system[1] running on TI Explorers.[2] POLYMER uses KEE's frame-based knowledge representation to encode domain activity, agent and object descriptions (see Section 2.1). KEE's *assumption-based truth maintenance system* (ATMS) is used to record all planning decisions and support dependency-directed backtracking (see Section 3.3). We have extended KEE's "world" system to permit the construction of a world hierarchy graph to represent the state of the application domain at each point in a POLYMER plan (see Section 3.2.2). Finally, KEE was selected to obtain an added degree of portability, especially in the design of POLYMER's interface.[3]

The overall architecture of the POLYMER system is presented in Figure 1. The domain representation and planning methodology are presented below. POLYMER's exception handling capabilities and knowledge presentation and acquisition facilities are discussed briefly in Section 4.

2.1 Representing the application domain

POLYMER models an application domain through the description of *activities*, *objects* and *agents* within the domain. In this section we focus on the description of domain

[1] KEE is a registered trademark of IntelliCorp, Inc.

[2] Explorer is a trademark of Texas Instruments Inc.

[3] A version of POLYMER has recently been ported to Sun (trademark of Sun Microsystems, Inc.) workstations by Olivetti.

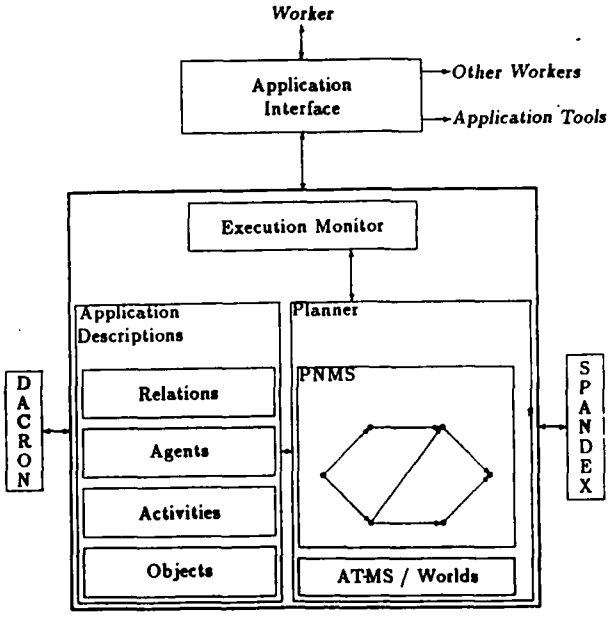


Figure 1: The POLYMER Planning System

activities. A grammar for the POLYMER activities appears in Appendix A; a sample activity description from journal editing is shown in Figure 2.

The representation includes both the *goal* and *preconditions* of an activity as well as a *decomposition* of the activity into smaller *steps*. The steps are usually goals for which other activities will be selected during the planning process, but may also be specific activities or tool invocations (i.e., "actions"). Causal relations between steps may be specified and these relations are used to generate temporal ordering constraints as well as protection intervals.[4] Control flow among the steps (e.g., looping, iteration, etc.) may also be specified directly.

Consider an example from the domain of journal editing. When a paper is submitted for publication in a journal, the editor of that journal must select appropriate reviewers for the paper, invite them to review the paper, collect their reviews, make a decision based on the reviews, and inform the author of the decision. Figure 2 shows a high-level POLYMER activity description for reviewing a submitted paper. It contains subgoals which are named and specified in terms of states of domain objects. The subgoals are causally (and therefore temporally) linked since a reviewer must be selected before a copy of the paper can be sent to the reviewer and the reviewer must receive the paper before the editor can get a review back. The two latter steps are repeated for each reviewer selected in the first step.

Using these domain descriptions, POLYMER interactively generates hierarchical, partially-ordered plans to accomplish a stated goal. The planning process, described below, is unique in its use of a combination of "script based" and "goal directed" de-

[4]Protection intervals maintain a certain state of the world during a portion of the plan on the assumption that a state generated at one point will be needed at some later point. See [20].

ACTIVITY: REVIEW-PAPER
Goal: reviews(?submission, ?reviews)
Preconditions:
member(?paper,papers)
edits.journal(?editor, ?journal)
submitted.to(?submisison, ?journal)
paper(?submission, ?paper)
Decomposition:
GOAL *reviewers-selected* =
and(reviewers(?submission, ?reviewers),
sufficient-reviewers(?journal,
?reviewers))
GOAL *paper-distributed* = has(?reviewer,
copy-of(?paper))
GOAL *have-review* = has(?editor, review(?reviewer))
Plan Rationale:
ENABLES reviewers-selected paper-distributed
ENABLES paper-distributed have-review
Control: **repeat** (paper-distributed **to**
have-review)
for ?reviewer **in** ?reviewers
Agents: ?(*editor*, editors)

Figure 2: A POLYMER Activity Description

scriptions of activities to overcome the rigidity of scripts while greatly reducing the cost of purely goal driven systems. It interleaves planning and plan execution in order to overcome the ambiguity inherent in complex, underspecified domains.

2.2 Preprocessing of domain descriptions

In order to make efficient use of the domain descriptions during the planning process, a certain amount of preprocessing is necessary. First, the external forms of the domain descriptions (shown above) are parsed and converted into Activity, Object, and Agent (KEE) units. Then, the activity descriptions are further processed to convert their information into a form more conveniently used during planning.

As we will see in the following section, POLYMER represents a plan as a *partially ordered network of plan nodes* called a *plan network*. To simplify the generation (and expansion) of this plan network, POLYMER creates a plan network to represent each activity during the preprocessing phase. Thus, all the information in an activity description is converted into a partially ordered set of plan nodes. A more formal description of

POLYMER's Plan Network Maintenance System appears in [3].

To generate a plan network for an activity, POLYMER first creates a plan node for each *step* in the activity's decomposition. *Goal*, *activity* and *action nodes* are generated for each corresponding step type. In addition, *structural* nodes are generated to represent the start and finish of the entire activity. Next, a partial ordering is established among the nodes (using the node's "predecessor" and "successor" fields) based on information in the activity's control clause. The more complex control constraints (e.g., *if*, *optional*, *star*, *plus* and *repeat*) result in the insertion of additional structural nodes.

Next, the causal relations in the plan rational clause are used to generate additional ordering constraints and protection intervals for the plan network. Finally, the activity's preconditions are transformed into constraints on the network's start-node, the activity's effects are placed on the network's finish-node, and any additional constraints specified in the activity description are placed on appropriate nodes within the network.

An example of the plan network generated for the activity description in Figure 2 is shown in Figure 3. A grammar for the specification of POLYMER plan networks and plan nodes appears in Appendix B.

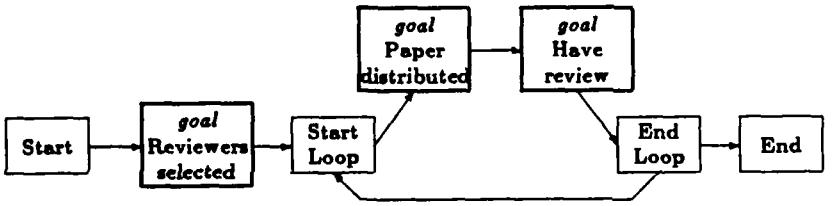


Figure 3: The "Referee Paper" Plan Network

In order to reduce the search for an activity to accomplish a goal during the planning process, one further preprocessing action is taken. For each goal node within an activity's plan network, POLYMER compares the value of the goal node to the goal of each known activity description. By finding and recording (during preprocessing) the set of all activities which could possibly satisfy each goal node, the planning process is made more efficient.

3 Interactive plan generation

To help a user achieve a desired goal, POLYMER attempts to generate a plan to accomplish the goal. To accomplish this, the goal and the required parameters must first be presented to the planner which then constructs the plan in a hierarchical manner. The planning proceeds in a top-down (in terms of plan abstraction), left-to-right (in terms of step ordering) fashion as far as possible without ambiguity. When the planning cannot proceed with certainty, it attempts to resolve the ambiguity by either 1) executing an action node (if any are "ready" as explained below) or 2) obtaining information from the user (such as which of several possible tasks it should use to accomplish an outstanding

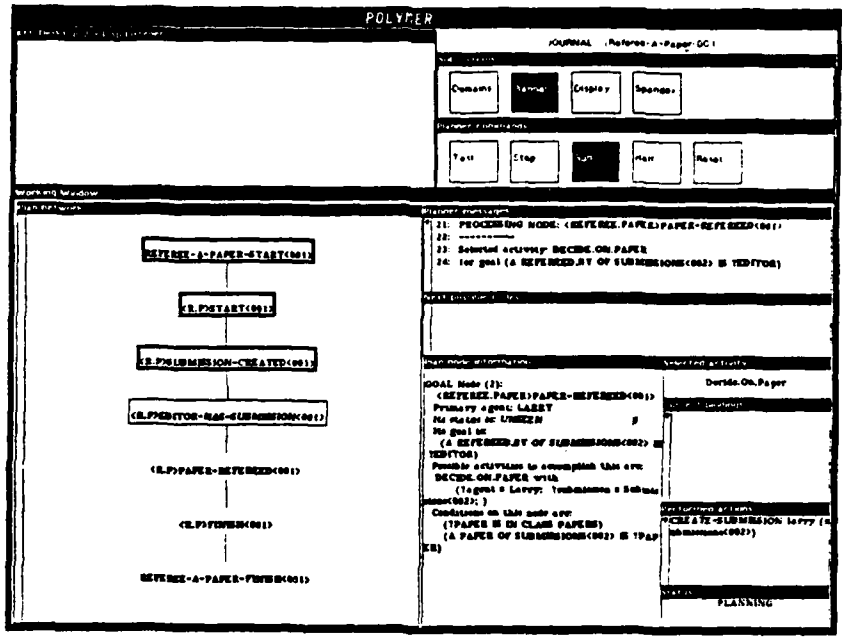


Figure 4: The POLYMER Developer's Interface

goal). In this section we describe how an initial goal is presented to the planner, how a plan network is interactively constructed to satisfy this goal, and what happens when difficulties arise during the generation and execution of the plan.

3.1 Statement of goal via an application interface

In a given application domain, there will typically be a fairly common set of goals that a particular user wishes to accomplish. For instance, in the journal editing domain, the editor of a journal will want to generate a request for papers, referee a submitted paper, modify a database of potential reviewers, etc. A reviewer will choose whether to review a particular paper and submit reviews to the editor. An author will write papers, submit them to journals, and rewrite the papers as necessary.

For a particular class of users in a particular domain, the user interface must allow the user to select the goal they want to accomplish and to specify the necessary parameters. Thus, a journal editor's interface allows the editor to state that he wishes to referee a paper and to specify the paper. The current system contains both a menu-driven "developer's" interface (Figure 4) and an iconic "end-user's" interface for the office domain (Figure 5).

Because each goal may require an extended period of time to be accomplished, a user will most likely want to interleave several tasks. Thus, the interface allows the user to suspend the current goal and select another (new or previously suspended) one.

Once a goal is selected, POLYMER generates a top-level plan network to represent that goal. It consists of a single goal node contained between a pair of "start" and

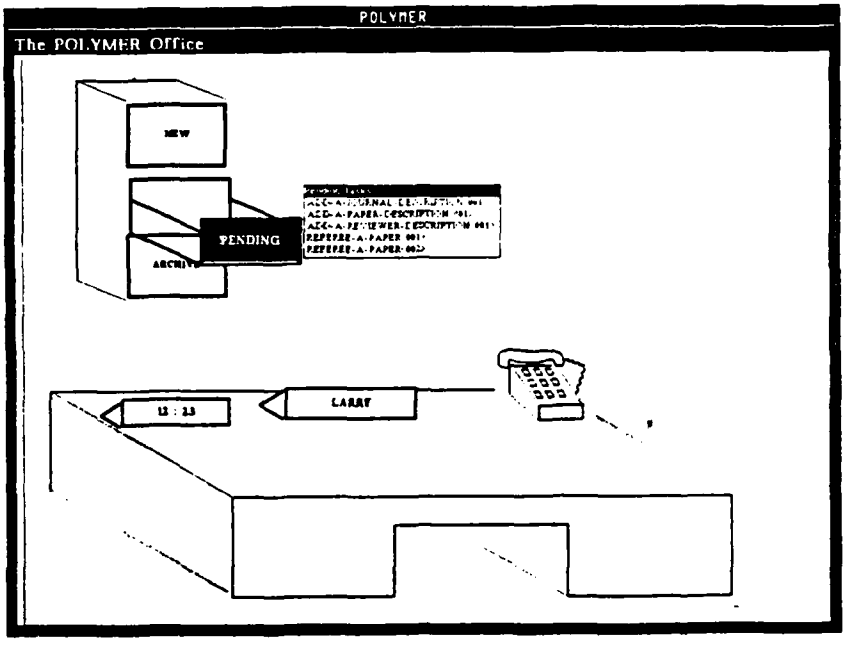


Figure 5: An Office Worker's Interface

"finish" structural nodes. The user's parameterized goal is used as the goal-value of the goal node; any initial state information is added to the top-level start-node. Planning begins by expanding this plan network as explained below.

3.2 Expanding the plan network

The POLYMER planner hierarchically constructs a plan by expanding a plan network. This expansion is performed by alternately selecting a plan node to process and then processing that node. The way in which a node is processed is determined by the node type.

3.2.1 Selecting a plan node

Because POLYMER constructs its plans in a top-down, left-to-right fashion, it selects the highest level (i.e., most abstract) node that is "ready" to be processed. In order to assure left-to-right processing, a node is "ready" to be processed if and only if 1) the node has not already been processed, 2) all of a node's necessary predecessors are complete and awaiting successors, and 3) all of the node's conditions are satisfied. A predecessor node is complete if its processing has been completed or if it does not need to be processed (i.e. a phantom goal node, as explained below).[5] A node's conditions are evaluated in

[5]A node's *join-type* determines whether it is necessary for all of the node's predecessors (for an "and" join) or merely any of them (for an "or" join) to be complete in order for the node to be "ready." Similarly, a node's *split-type* determines whether it is awaiting a successor: A node is awaiting successors until all of its successors are processed if it is an "and" split, or until any of its successors are processed if it is an "or" split.

the node's "before-world"[6] to assure that the node is applicable at this point in the plan.

To ensure top-down processing, each node is assigned an abstraction level denoting its "depth" from the original top-level goal. Thus, the nodes in the top-level network are assigned a level of 0; when a goal node is replaced by an activity network (as explained below), the level of the new nodes is 1 greater than the goal node they replace.[7]

Once the "ready" nodes are sorted by abstraction level, they are ordered by node type. Structural nodes (except for loop-iteration nodes) are processed first, followed in order by goal nodes, activity nodes, action nodes and finally loop-iteration nodes. If more than one node of a given type is ready to be processed, the planner can either select one based on a (modifiable) set of heuristics (e.g., specificity of the node's conditions, a priori preferences among tasks, etc.) or ask the user to select which node to process next.

3.2.2 Processing a plan node

Once POLYMER selects a plan node, the way in which the node is processed depends upon its type. For instance, it checks whether goal nodes are already satisfied by evaluating the goal node's value in the node's before-world. If the goal is already satisfied (either accomplished by some earlier steps or true in the initial world state), the node status is set to "phantom" and no further processing of the node takes place. If the goal is not satisfied, POLYMER determines if it can add additional ordering constraints to the existing plan network in order to place the node where its goal will be satisfied. If it is unable to achieve this, the planner must select an activity to achieve the goal.

As described in Section 2.2 above, each goal node template contains a list of all activities which may possibly satisfy the goal. These activities are now checked to see if their goals match that of the *instantiated* goal node and whether their preconditions are satisfied in the goal node's before-world. If more than one activity still qualifies as a means of accomplishing the goal, the planner can select one heuristically (using, for example, the closeness of the match between the activity's goal and that of the goal node, the specificity of the activity's preconditions, etc.) or by asking the user which activity should be performed.

Once an activity is selected for a goal, the plan network for that activity is instantiated and spliced into the current plan network in place of the goal node. Instantiating the network includes the instantiation of each of the nodes in the network, creation of KEE worlds corresponding to the new nodes, assertion of each node's effects in its after-world, and instantiating any needed protection intervals on these worlds. In addition to splicing the new nodes into the existing plan network, the new worlds are spliced into the existing world hierarchy.

Activity nodes are processed in a similar way, except that the selection of an appropriate activity is obviated. Action nodes require the invocation of tools (or interactions

---

[6]The "world" corresponding to the point in the plan immediately before the plan node is called its *before-world*; its *after-world* occurs immediately after the node.

[7]Therefore, nodes which are "higher" in terms of abstraction have "lower" level numbers.

with other agents) and are handled by the *execution monitor*. The tools are invoked as specified in the "code" portion of the action node and the results are recorded in the after-world corresponding to the action node. Note that action nodes can be processed before the plan network is completely expanded. This permits the interleaving of planning and plan execution in order to prevent the planner from becoming swamped by the potentially explosive combinatorics of purely strategic planning in an underspecified and often ambiguous environment.

Most structural nodes simply serve as a means of demarcating activity boundaries and are the appropriate locations to place constraints and effects that belong at the beginning or end of an activity. Thus, an activity's *preconditions* and *effects* are placed on the activity's start and finish-nodes, respectively. The processing of these nodes only requires checking that their conditions are valid in order for them to be "complete." Structural nodes used to control looping, generated from certain control constructs, are more complex. For each loop construct, a *loop-controller* object is created and the loop-iteration and loop-termination nodes have conditions and effects which utilize the loop-controller. Thus, if the conditions of a loop-iteration node are satisfied, an additional instantiation of the nodes which comprise the body of the loop are created and inserted into the plan network. If a loop-termination node is satisfied, both the loop-iteration node and the loop-termination node are marked "complete" and the looping terminates.

3.3 Detection and correction of plan problems

The expansion of the plan network and the execution of actions interleaved with plan generation can both cause problems to arise in the plan. These may be detected as 1) actions performed which were not expected as part of the plan, 2) goals selected by a user which were not currently expected, or 3) inconsistencies in the world model arising from violated protection intervals or failed constraints.

These problems can indicate an error on the part of the planner, "exceptional" behavior by the user, or simply a user error. The handling of exceptional behavior (i.e., intentional actions by the user that are not covered by the planner's domain model) is described in Section 4.1. In order to correct errors in the existing plan, POLYMER first considers the addition of node ordering constraints to resolve the problem. If this fails and the problem was caused by a violated goal, it considers reinstating the goal at a point after it was violated and before it is needed.

If POLYMER is unable to resolve the problem by manipulating the existing plan network, it is forced to backtrack and undo some of its previous planning decisions (e.g., the selection of an activity for a goal, a choice of alternative goals, etc.). Because POLYMER uses KEE's ATMS to justify and record each of its planning decisions, the planner needs to redo only those portions of the plan that led to the problematic results (i.e., dependency-directed backtracking).

4 Beyond the basic planner

While we believe that a planner such as POLYMER is an essential component to support work in cooperative environments, we also see the planner as the core of a set of sophisticated support tools. Several research projects are currently underway that build upon POLYMER's functionality and aim to extend its overall utility. These projects, described below, include systems to handle exceptional behavior, to present and acquire models of application domains, and to support conflict resolution.

4.1 Exception handling

Because POLYMER's domain model is inherently incomplete (as is *any* model of a "real-world" domain), there will be situations where the planner does not correctly anticipate a user's desired action(s). By combining the domain model with heuristic knowledge about how plans may deviate, the SPANDEX system [4] uses a process of *plausible inference* to generate *explanations* of how a user's exceptional behavior can be reconciled with an existing POLYMER plan. Using these explanations, SPANDEX constructs the necessary *amendments* to the domain model to incorporate this new behavior.

4.2 Knowledge presentation and acquisition

Though POLYMER's domain model may never be complete (or even necessarily correct), the ability to make modifications (e.g., additions, corrections) in a simple fashion is extremely important. In order to make such modifications *by end users* feasible, the domain model must be presented to and manipulated by the users in an "understandable" fashion. The DACRON project [15] has investigated how typical end-users perceive tasks and objects within their domains and has been able to map this more "natural" model onto the POLYMER formalism. An interactive, animated, iconic interface is being developed to permit the presentation of information to these users and allow them to modify existing information as well as to specify additional tasks, objects, etc.

4.3 Conflict resolution

Because cooperative work environments requires the interaction of multiple agents (as well as the planner), conflicts between agents will inevitably arise. Differing goals, limited knowledge about the domain, varying capabilities and incomplete models of other workers can lead one agent to perform in a way that another agent does not expect. The *resolution* of such conflicts is often difficult and usually occurs through negotiation between the affected agents[2]. The GENEVA project [8] has begun to explore ways in which POLYMER's models of the domain and of the current plan can be used to *support* the negotiation process. In particular, it aims to assist in 1) initiating negotiation (by identifying the needed agents and presenting them with an appropriate view of the conflict), 2) maintaining the state of current and past negotiation sessions, 3) suggesting and

allowing the exploration of solutions, and 4) verifying that proposed solutions actually resolve the conflict.

5 Summary

The POLYMER planning system has been designed and implemented as the core of an environment to support cooperative work. The current prototype has been used to interactively generate plans in such diverse areas as journal editing, software development and house purchasing. A preliminary version of the system has been delivered to an Olivetti research laboratory where it is being used to develop advanced applications in the area of office automation.

In addition to the development of further applications, POLYMER is serving as a testbed for several research projects. These projects are exploring the use of knowledge acquisition, exception handling, and computer-mediated conflict resolution as part of an effort to develop an integrated environment for the support of cooperative work.

References

[1] G.R. Barber, "Supporting Organizational Problem Solving with a Work Station," *ACM Transactions on Office Information Systems*, 1, pp. 45–67, 1983.

[2] O.J. Bartos, "Process and Outcome of Negotiation," Columbia University Press, New York, 1974.

[3] M. Beetz and L.S. Lefkowitz, "Efficient Planning in Dynamic Multiagent Domains," Technical Report, COINS Department, University of Massachusetts, Amherst, Massachusetts, 1988.

[4] C. Broverman, and W.B. Croft, "Reasoning about Exceptions during Plan Execution Monitoring," *Proceedings of the AAAI-87*, 1987.

[5] D. Chapman, "Planning for Conjunctive Goals," *Artificial Intelligence*, 32, pp. 333–377, 1987.

[6] W.B. Croft and L.S. Lefkowitz, "A Goal-based Representation of Office Work," *Proceedings of the IFIP Conference on Office Knowledge*, 1988. (Also in *Office Knowledge: Representation, Management, and Utilization*, North Holland, 1988.)

[7] W.B. Croft and L.S. Lefkowitz, "Knowledge-based Support of Cooperative Activities," *Proceedings of the 21st Hawaii International Conference on System Sciences*, January 1988. (Also in *Readings on Distributed AI*, Morgan Kaufmann, 1988.)

[8] W.B. Croft and L.S. Lefkowitz, "Computer-Mediated Conflict Resolution," Technical Report, COINS Department, University of Massachusetts, Amherst, Massachusetts, May 1988.

[9] W.B. Croft and L.S. Lefkowitz, "Task Support in an Office System," *ACM Transactions on Office Information Systems*, Vol. 2, July 1984.

[10] W.B. Croft and L.S. Lefkowitz, "Using a Planner to Support Office Work," *Proceedings of the ACM Conference on Office Information Systems*, March 1988.

[11] R.E. Fikes, "A Commitment-based Framework for Describing Informal Cooperative Work," *Cognitive Science*, 6, pp. 331–347, 1982.

[12] R.E. Fikes and D.A. Henderson, "On Supporting the Use of Procedures in Office Work," *Proceedings of the AAAI–80*, 1980.

[13] M.P. Georgeff, "Reasoning about Plans and Actions," *Exploring Artificial Intelligence*, H. Shrobe (editor), pp. 173–196, Morgan Kaufmann, 1988.

[14] B. Grosz and C. Sidner, "Distributed Know-How and Acting: Research on Collaborative Planning," *Proceedings of the DARPA DAI Workshop*, 1988.

[15] D.E. Mahling and W.B. Croft, "An Interface for the Specification of Office Activities," *Proceedings of the IFIP Conference on Office Information Systems*, Linz, Austria, August 1988.

[16] T.W. Malone, "What is Coordination Theory?" presented at the *NSF Coordination Theory Workshop*, 1988.

[17] A. Sathi, T.E. Morton, and S.F. Roth, "Callisto: An Intelligent Project Management System," *AI Magazine* Vol. 7, No. 5., 1986.

[18] W. Swartout (editor), "DARPA Santa Cruz Workshop on Planning," *AI Magazine* Vol. 9, No. 2., 1988.

[19] A. Tate, "Generating Project Networks," *Proceedings IJCAI–77*, pp. 888–893, 1977.

[20] D.E. Wilkins, "Recovering from Execution Errors in SIPE," *SRI Technical Report 346*, 1985.

A A Grammar for Activities, Objects, and Agents

activity	:=	**ACTIVITY** *activity-name* activity-clause*
activity-clause	:=	goal-clause # [preconditions-clause] # [effects-clause] # [decomposition-clause] # [rationale-clause] # [control-clause] # [agents-clause] # [constraints-clause]
goal-clause	:=	**GOAL** world-predicate
preconditions-clause	:=	**PRECONDITIONS** world-state*
effects-clause	:=	**EFFECTS** effect-spec*
decomposition-clause	:=	**DECOMPOSITION** step*
rationale-clause	:=	**PLAN-RATIONALE** enables-relation*
control-clause	:=	**CONTROL** control-construct*
agents-clause	:=	**AGENTS** agent-spec*
constraints-clause	:=	**CONSTRAINTS** world-state*
step	:=	step-spec [**done-by** agent-spec]
step-spec	:=	(**goal** *step-name* world-state) | (**activity** *step-name* {*activity-name* | (**one-of** *activity-name**)}) | (**action** *step-name action-name* parameter-list [world-state])
agent-spec	:=	world-state | kb-entity
parameter-list	:=	({variable, }* variable)
control-construct	:=	**before**{*step-name*, }+ *step-name* | **if** world-state **then** step-or-net [**else** step-or-net] | **optional** step-or-net | **star** step-or-net | **plus** step-or-net | **repeat** step-or-net repeat-bounds [**iterate-when** world-state]
repeat-bounds	:=	**while** world-state | **until** world-state | **times** *integer* | **for** variable **in** ({value, }* value) | **with** variable **suchthat** world-state
step-or-net	:=	*step-name* | (*step-name* **to** *step-name*)
enables-relation	:=	**enables**({*step-name*, }+ *step-name*)
effect-spec	:=	(effect-action world-state)
effect-action	:=	**set** | **add** | **delete**
world-state	:=	predicate (kb-term, kb-term) | **not**(world-state) | **and**({world-state, }* world-state) | **or**({world-state, }* world-state)
kb-term	:=	predicate (kb-term) | *kb-entity* | value | variable
predicate	:=	system-predicate | *kb-predicate*
system-predicate	:=	**member** | **subclass** | **equal**
value	:=	*string* | *number* | symbol-not-a-var
variable	:=	unconstrained-variable | constrained-variable
unconstrained-variable	:=	`any symbol whose first character is a "?"`
constrained-variable	:=	?(symbol-not-a-var {world-state | *kb-entity*})
symbol-not-a-var	:=	`any symbol whose first character is not a "?"`

B A Grammar for Plan Networks and Nodes

plan-network := **PLAN-NETWORK** *network-name*
start-node # finish-node # from-activity

start-node := (**start** plan-node)

finish-node := (**finish** plan-node)

from-activity := (**from-activity** *activity-name*)

plan-node := **PLAN-NODE** *node-name* node-info

node-info := goal-node | activity-node | action-node | structural-node

goal-node := generic-node # (**type goal**) # (**goal** world-state) #
(**possible-activities** activities-and-bindings) #
(**selected-activity** *activity-name*) #
(**expansion-network** plan-network)

activity-node := generic-node # (**type activity**) # (**goal** world-state) #
(**selected-activity** *activity-name*)

action-node := generic-node # (**type action**) # (**goal** [world-state]) #
(**code** *fn-name*)

structural-node := generic-node # (**type structural**) # (**loop-info** loop-controller)

generic-node := (**predecessors** (*node-name**)) # (**successors** (*node-name**)) #
(**from-node** *node-name*) #(**from-activity** *activity-name*) #
(**before-world** *world-name*) #(**after-world** *world-name*) #
(**start-time** time-range) # (**finish-time** time-range) #
(**split-type** {**and** | **or**}) #(**join-type** {**and** | **or**}) #
(**level** *number*) # (**step-name** *step-name*) # (**agent** agent-spec) #
(**status** { **unseen** | **expanded** | **pending** | **phantom** |
complete | **looping** }) #
(**repeat-from** (*node-name**)) #(**repeat-after** (*node-name**)) #
(**conditions** world-state*) #(**if-bound-conditions** world-state*) #
(**effects** effect-spec*)

activities-and-bindings := (activity-and-bindings*)

activity-and-bindings := (*activity-name* bindings-list)

bindings-list := ((variable kb-term)*)

loop-controller := **LOOP-CONTROLLER** *controller-name* loop-controller-info

loop-controller-info := (**initial-list** (kb-term*)) # (**initial-value** kb-term) #
(**current-list** (kb-term*)) # (**current-value** kb-term) #
(**variable** variable)

time-range := (time-spec, time-spec)

time-spec := *number* [time-unit]

time-unit := **seconds** | **minutes** | **hours** | **days** | **weeks** | **years**

Appendix 6-D

Plausible Explanations to Cope with Unanticipated Behavior in Planning

Carol A. Broverman and W. Bruce Croft

Abstract: Complex tasks can be accomplished efficiently by human agents with the assistance of a hierarchical nonlinear planner. However, since such a paradigm implies an active role of the human agents in making decisions and providing information, an interactive interface must be prepared to encounter unusual behavior which deviates from system expectations. This paper describes a system (SPANDEX) which constructs *plausible explanations* as justifications for agent behavior. The verification of an explanation can involve a reorganization of existing knowledge or require the acquisition of additional knowledge. Selected explanations are implemented through proposed *amendments* to the knowledge base. Detailed examples of the operation of the system are presented.

This work is supported by the Air Force Systems Command, Rome Air Development Center, Griffiss Air Force Base, New York 13441-5700, the Air Force Office of Scientific Research, Bolling Air Force Base, District of Columbia 20332, under contract F30602-85-C-0008, and by a contract with Ing. C. Olivetti & C.

1 Introduction

No matter how carefully a plan is conceived, things frequently go wrong during its execution. When human agents are responsible for executing plan steps this problem is of particular importance. People change their minds or opportunistically revise a plan mid-execution. In addition, they are prone to error and misjudgement. Consequently, a system designed to support the performance of human tasks in an interactive setting [4,5] must be prepared to cope with frequent unanticipated occurrences (*exceptions* [2]) during the planning and execution process.

In [2], we describe an interactive planning system and its requirements for a general exception handling mechanism. In this setting, the input of human agents is required for task completion and thus exceptions can be generated by the actions of known agents. In particular, an agent may perform an action which is inconsistent with system predictions. For example, an agent may leave out a step in a task as a deliberate short cut, or he may perform an unexpected action as an intentional substitution of an expected action. In other cases, the action of an agent may not be an intentional aberration, but may be viewed initially as an exception due to an incomplete or incorrect domain plan library.

Such exceptions are referred to as *accountable* since it is presumed that *agents behave purposefully* and there are *motivations*[1] for their behavior. Actions which initially appear to be "errors" can often be recognized and explained as actions consistent with the goals of the plan. We contrast this class of accountable exceptions with the more frequently addressed arbitrary changes in world state brought about by unknown agents. For example, a system that is planning a travel itinerary may have to contend with the effects of an earthquake which has forced the cancellation of a scheduled train. We refer to this latter type of exception as *unaccountable* and recognize replanning as an effective approach for plan recovery [11,15].

Accountable exceptions, however, should be justified rather than "counteracted" through replanning. Explanations of unanticipated agent behavior can result in improvements in both system understanding and performance. As an initial step towards achieving this aim, we have defined the categories of accountable exceptions that can arise in a cooperative planning framework. The types of exceptions defined by this behavioral perspective are: *action not in plan, out of order action, repeated action, user assertion*, and *expected action with parameter causing constraint violation.* A complete description of this taxonomy and additional groundwork for the approach described in this paper can be found in [3].

In this paper we investigate the construction of *plausible explanations* of accountable

[1]What constitutes a relevant agent motivation may vary from one domain to the next, since the policies and constraints of the work setting are influential. For example, an environment with strong financial incentives may encourage individuals to act in ways to cut costs even at the expense of lengthy tasks, while in other settings financial expense may be considered as secondary to task simplification. Other possible motivations for deviating from expected procedure include: anticipation, partial achievement of an expected plan step, and special case handling.

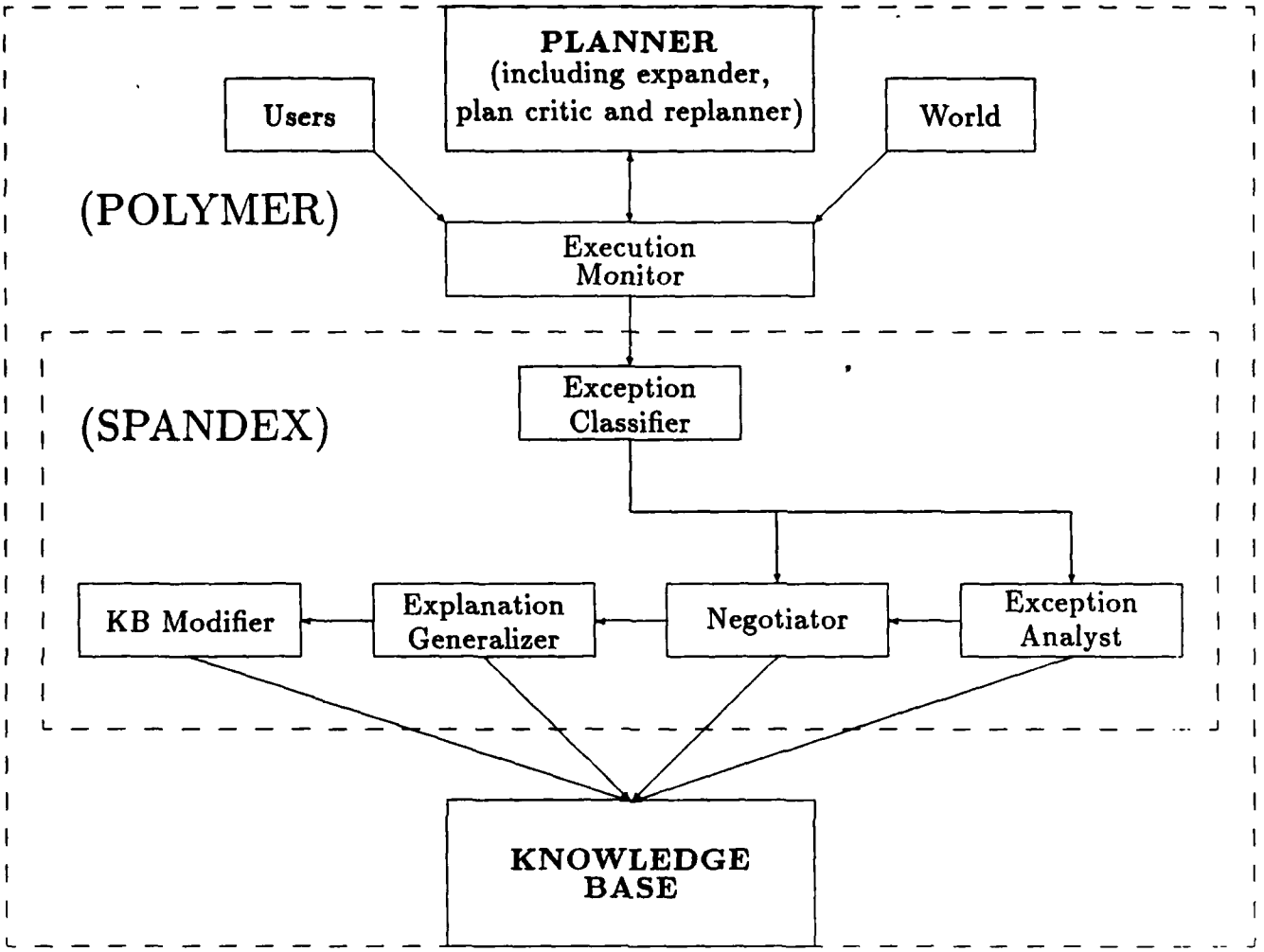


Figure 1: Architecture for a cooperative planning system

exceptions. Section 2 describes the implemented system SPANDEX (Support for Planning and EXception handling) and its interface to POLYMER, a hierarchical planning system similar to NOAH [12] and NONLIN [14]. SPANDEX allows a planning system to continue the planning and execution of a task after encountering an exception. The approach described here facilitates the extension of an existing knowledge base to accommodate alternative ways to complete task goals, as learned through the handling of previous exceptions. In section 3, we introduce the concept of *plausible explanations* as justifications of presumed agent motives and describe how they are constructed. Section 4 discusses how *amendments* to the knowledge base are proposed to restore consistency to the plan network upon acceptance of an explanation. An example of the operation of SPANDEX in the domain of journal editing is presented throughout for illustration, and an additional example from the software development domain is given in the Appendix.

2 An architecture to support exception handling

POLYMER [5,6] is an interactive planning system designed to assist in the management of tasks in a cooperative setting. It uses a hierarchical planner to construct a procedural net that specifies the sequences of actions required to achieve a goal. POLYMER constructs partial plans and executes them in cooperation with agents. The actual actions taken by these agents are compared to expected actions, and when differences are found (producing an exception) SPANDEX is invoked. This architecture is shown in Figure 1. In this section, we describe the SPANDEX architecture and introduce an example to illustrate the mechanisms described.

2.1 The SPANDEX architecture

When an exception is detected by the plan execution monitor, the *exception classifier* is invoked to determine the exception type. The *plan critic* determines if goal nodes have been violated in the plan as a result of the exception. The *replanner* handles unaccountable exceptions (generated by *world* in Figure 1.

The *exception analyst* applies domain knowledge to construct *plausible explanations* of accountable exceptions. The function of the exception analyst is described in more detail in [2,3] and plausible explanations are discussed further in the following section. Since several agents can be affected by an exception, we propose to use negotiation [10,13] to establish a consensus among them regarding the explanation and proposed plan modifications. The *negotiator* identifies the affected agents and uses the information provided by the exception analyst to conduct a dialogue. The output of the negotiation phase is a selected explanation, along with approved changes to be made to either the plan network for this particular instance or to the permanent plan library.

The *explanation generalizer* produces a generalized form of the verified explanation, using taxonomic information in the knowledge base. This new knowledge about domain activities, along with any suggested knowledge base changes resulting from negotiation, is passed to the *knowledge base modifier*. Thus, a successful negotiation can result in a system which has "learned," that is, the static domain plans may be augmented with knowledge about the exception and thus the system is able to handle future similar exceptions.

2.2 An example

To illustrate the mechanisms discussed in this paper, we will develop an example involving a journal editing task. The goal of the task is to decide whether or not to accept a paper for publication. The primary agent initiating the task is the editor, who has received a paper for review in the area of artificial intelligence. POLYMER generates a partial plan network for this task, and executes it in conjunction with the relevant agents. There are three ordered subgoals generated for this task: *reviewers-selected* (the

Unit-name: DYNAMIC.OBJECT.CONSTRAINT.VIOLATION13
Unit-comment: "A constraint dynamically posted on a KB object has been violated. "
Exceptional-action: select-reviewer1
Exception-summary:
"The target constraint: *(area.of.expertise Seymour.Wright artificial.intelligence)*
was not met; it is the case that *(area.of.expertise Seymour.Wright computer.vision)*."
Target.constraint:*(area.of.expertise Seymour.Wright artificial.intelligence)*
Perceived.constraint *(area.of.expertise Seymour.Wright computer.vision)*
Strategies: dynamic.object.constraint.strategy
Strategy.selector: strategy.selector.method
Explanations: specialisation.constraint.values.expl14, generalisation.constraint.values.expl15

Figure 2: *Dynamic.object.constraint.violation13*

editor must select reviewers to judge the submission), *reviews-received* (the actual reviews must be obtained), and *decision-reached* (the editor must make a final decision based on the responses). A constraint on the *reviewers-selected* task subgoal specifies that each reviewer selected must have an area of expertise which matches the area of the paper to be reviewed.

The agent (editor) is instructed to perform the activity *select-reviewer* to achieve the first subgoal. The activity is performed and the reviewer selected is *Seymour.Wright*. When the constraint is evaluated, it is discovered that the area of expertise of *Seymour.Wright* is *computer.vision* rather than *artificial.intelligence*. SPANDEX is invoked by POLYMER upon detection of this exception, and the exception classifier determines that a constraint which was dynamically posted on a knowledge base object has been violated (a *dynamic object constraint violation* exception). The *exception record* pictured in Figure 2 is generated to summarize the exception. The exception classifier first records the action which generated the exception, and the type of exception which was detected. The actual constraint which was violated (*target.constraint*) is recorded, and SPANDEX also determines the actual value in the knowledge base object that triggered the constraint violation (contained in *perceived.constraint*). SPANDEX then invokes the exception analyst, which uses the *strategy.selector* to choose one of the *strategies* to generate *explanations* for the exception.

In the next section, we describe how explanations are generated for exceptions, and illustrate the behavior of the exception analyst on the example introduced above.

3 Plausible explanations

The term *explanation* is broadly used in current artificial intelligent research. In *explanation-based learning* [7,8,9], explanations are generated as *proof* that a sequence of steps achieves its goal. This type of explanation is logically sound, and is in effect a restructur-

ing of existing knowledge. In POLYMER we also would like to construct explanations which justify how exceptions contribute toward current plan goals. Our approach, however, is not to supply a rigorous proof, but rather to suggest plausible roles for the exception in the context of the plan. Due to the interactive nature of our planner, we often have only a partial action sequence available for analysis, and assume a potentially incomplete knowledge base. Therefore, we are not always able to produce formal explanations of exceptions through the reorganization of existing knowledge, but must rely on the construction of potentially valid (*plausible*) explanations which may require user validation and additional knowledge for verification.

The task of the exception analyst is to apply a set of algorithms to the knowledge base in order to produce plausible explanations of exceptional behavior. The algorithms are based on *plausible inference rules* and conduct a controlled exploration of a rich and integrated representation of domain activities and objects [2]. In this section, we define what we mean by explanations, and show how they are constructed, using the example introduced in the previous section.

3.1 Plausible inference rules

For a given exception type, a set of plausible inference rules are retrieved. These rules are intended to reflect possible motivations of the responsible agents and are used to construct explanations for the exception. The general form of a plausible inference rule is a set of conditions (forming the premise of the rule) followed by the established rationale for allowing the exception (conclusion). For example, one plausible inference rule used during the resolution of an exceptional action resulting in a constraint violation is the following:

> **Plausible-inference-rule1:** If the violating value in a constraint predicate is a specialization of the target value of the violated constraint, then the violating value may suffice as a substitution for the target value.

In this case, we have a single condition in the premise (specifying a taxonomic relationship that must exist between actual and target values in a constraint), but in the general case there may be several conditions. Each condition specifies a semantic relationship which must hold among one or more entities in the knowledge base. These relationships include: direct specialization, direct generalization, sibling or cousin taxonomic relationships, and causal relationships. These and other semantic relationships are discussed in more detail in [3]. Parameters are associated with some of the condition types; for example, when a specialization relationship is established in an explanation, the number of taxonomic links between the two objects is recorded (a measure of closeness), or for an established generalization relationship, it may be the case that the more specific object has five additional attributes (a measure of similarity). The values of the condition parameters establish the *degree of plausibility* of each explanation to enable the ranking of multiple explanations.

3.2 Explanations

Each plausible inference rule retrieved for an exception gives rise to the instance of an *explanation*. Therefore, there may be many possible explanations for a given exception. An explanation is considered *complete* if all of the conditions in the premise can be substantiated by SPANDEX. If one or more of the conditions cannot be verified, the explanation is *incomplete*. Complete explanations are a result of analyzing the existing knowledge to infer information from existing facts. Incomplete explanations, on the other hand, represent lines of reasoning that may result in valid explanations of an exception if we are able to verify the missing conditions. In other words, the construction and acceptance of a complete explanation involves a *reorganization* of existing knowledge[2], while incomplete explanations require the *acquisition of new knowledge* in order to be substantiated and accepted. Complete explanations are preferred since they are already verified and can allow the successful completion of a plan in progress, but incomplete explanations can potentially lead to improved system performance through the acquisition of new knowledge.

When an explanation is considered to be incomplete, an attempt to complete it may be warranted for one of two reasons. First, the knowledge base may be incomplete; an unverified relationship or fact may be simply *unknown*, and might be acquired through a dialogue with a responsible agent. Secondly, the knowledge base may be incorrect; a condition in an explanation which is false could be established as true through a dialogue with an agent. Thus, the negotiation involving incomplete explanations plays a primary role in both the expansion and debugging of the initial knowledge base.

In the example introduced in section 2.2, SPANDEX constructs two plausible explanations, one which is complete, and one which is incomplete (see Figure 3). In this particular case, the complete explanation states that since the referee selected has an *area.of.expertise* which is a specialization of the required *area.of.expertise*, he is sufficiently qualified to review the paper of concern. Note that since the specialization relationship (*computer.vision* is a subclass of *artificial.intelligence*) exists in the knowledge base, *generalization.constraint.values.expl14* is not only incomplete but *invalid*, since an object cannot be both a generalization and a specialization of the same object. To illustrate the handling of incomplete explanations suppose, however, that the taxonomic link between *computer.vision* and *artificial intelligence* was not initially specified in the knowledge base. If this had been the case, the two explanations in Figure 3 would have been constructed as *incomplete* (but potentially valid) explanations. Either one of them may have been chosen during the negotiation process and completed by verifying with a knowledgeable agent that the appropriate missing taxonomic link could indeed be added to the knowledge base.

[2]Note that complete explanations are similar to those explanations produced by the explanation-based learning paradigm discussed earlier.

Unit-name: SPECIALIZATION.CONSTRAINT.VALUES.EXPL15
Unit-comment: "Show that the actual value of the violated constraint predicate is a specialisation of the target value in the constraint."
Explanation-summary: "The actual value of the *area.of.expertise* field of *Seymour.Wright* (*computer.vision*) is sufficient since it is a specialisation of the desired value *artificial.intelligence*."
Status: complete
Reasoning.method: specialisation.constraint.values
Hierarchy.level.difference: 1

Unit-name: GENERALIZATION.CONSTRAINT.VALUES.EXPL14
Unit-comment: "Show that the actual value of the violated constraint predicate is a generalisation of the target value in the constraint."
Explanation-summary: "If the value of the *area.of.expertise* field of *Seymour.Wright* (*computer.vision*) had been a generalisation of the target constraint value *artificial.intelligence*, this explanation would be valid."
Status: incomplete
Reasoning.method: generalisation.constraint.values

Figure 3: Explanations for *Dynamic.object.constraint.violation13*

4 Amendments

Once the candidate explanations for an exception have been generated by the exception analyzer, a selection must be made. Currently, summaries of the candidate explanations are presented to the user, who makes a choice. Once the selection of an explanation has been made, it must be *implemented* to restore system consistency and enable the resumption of execution. The implementation of an explanation is specified by one or more *amendments* to be made to the static or dynamic state of the system. At this time, the choice of the amendment is also made by the user, although this choice could be automated by weighing the implementation costs of the candidate amendments.

Each plausible inference rule has one or more amendment types associated with it, based on the conditions involved in its premise. Amendments are generated only for complete explanations, unless an incomplete explanation is chosen and verified through agent interaction. In this paper, we will describe the concept of amendments by discussing the subset of exceptions which generate constraint violations, and illustrate with the amendments (see Figure 4) generated for the complete explanation *specialization.constraint.values.expl15*.

In general, when encountering a constraint violation, there are three fundamental approaches to resolving the problem[3]:

1. Relax the constraint. Possible ways to do this are:

[3]The first approach is largely based on work described in [1].

Unit-name: CONSTRAINT.VALUE.DISJUNCT.ADDITION16
Unit-comment: "Modify a current.constraint by adding an additional.value
in a disjunct.clause to produce a new.constraint."
Amendment-summary: "Replace the current constraint
(area.of.expertise Seymour.Wright artificial.intelligence) with the new constraint
(or (area.of.expertise Seymour.Wright artificial.intelligence)
(area.of.expertise Seymour.Wright computer.vision))."
Implementation: (add-values-to-constraint (computer.vision)
(area.of.expertise Seymour.Wright artificial.intelligence))

Unit-name: CONSTRAINT.VALUE.TAXONOMIC.EXPANSION17
Unit-comment: "Modify a current.constraint by replacing the existing.value with
its taxonomic.expansion to produce a new.constraint."
Amendment-summary: "Replace the current constraint
(area.of.expertise Seymour.Wright artificial.intelligence) with the new constraint
(area.of.expertise Seymour.Wright
(or artificial.intelligence distributed.ai planning robotics computer.vision))."
Implementation: (replace-values-in-constraint
(artificial.intelligence distributed.ai planning robotics computer.vision)
(area.of.expertise Seymour.Wright artificial.intelligence))

Unit-name: TARGET.OBJECT.ATTRIBUTE.VALUE.ADDITION18
Unit-comment: "Modify a KB object by adding a new.value to
the existing.values in an object.attribute field of the object."
Amendment-summary: "Add the new value *artificial.intelligence* to the current values
(*computer.vision*) of the *area.of.expertise* field of the KB object *Seymour.Wright*."
Implementation: (add-attribute-values-to-kb-object (artificial.intelligence)
area.of.expertise Seymour.Wright)

Figure 4: Amendments for *Specialization.constraint.values.expl15*

(a) Disjunct addition (add a disjunct to the specification);

(b) Conjunct elimination (eliminate one or more conjuncts from the specification);

(c) Taxonomic generalization (replace a class specification with a more general superclass);

(d) Taxonomic expansion (replace a class specification by a set of classes which are subsumed by the original class specification);

(e) Range extension (extend a numeric or other ordinal range);

(f) Constraint elimination (eliminate the constraint);

2. Remove the source of constraint violation.

   (a) The constraint may have failed because of unknown information. Perhaps the missing knowledge can be acquired, resulting in a successful evaluation of the constraint.

   (b) Information in the knowledge base may be incorrect. A change should be made to the knowledge base so that the subsequent evaluation of the constraint is successful.

3. Undo what was done, and do it differently (replan).

Conceptually, the first approach implies that a constraint was specified incorrectly; it *was too strict. The second appoach implies* that some knowledge is either not explicit or is incorrect in the knowledge base, and that additional information can either be inferred or acquired in order to nullify the violation. With these concepts in mind, the amendments shown in Figure 4 were produced for the example exception summarized in Figure 2. Note that the first two proposed amendments illustrate the first approach. They use the techniques of disjunct addition and taxonomic expansion to actually change the constraint specification to accommodate the exception. The third amendment is based on the second approach, and involves making an actual change to a knowledge base domain object. Thus, the actual source of the constraint violation is removed. The user selects one of these amendments to indicate how the explanation should be implemented. The knowledge base modifier performs the indicated changes specified in the *implementation* field of the amendment, and planning and execution by POLYMER is resumed.

5 Status

A prototype system implementing the SPANDEX architecture is integrated with the POLYMER planner and running on a Texas Instruments Explorer. The exception analyzer currently handles a subset of the exception types; algorithms for the remainder are being implemented. Generalization techniques such as those described in [1] are being examined as the basis for the explanation generalizer. The negotiator is not yet implemented.

Unit-name: ACTION.MISMATCH.01
Unit-comment: "The action taken by an agent did not match the action expected by the planner. "
Exceptional-action: unix-make-1
Exception-summary:
"The target action: *compile-file-01* did not occur,
it is the case that *unix-make-01* was performed."
Target.action: *compile-file-01*
Perceived.action *unix-make-01*
Strategies: action.substitution.strategy, out.of.order.action.strategy
Strategy.selector: strategy.selector.method
Explanations: substitute.for.higher.level.goal.01

Figure 5: ACTION.MISMATCH.01

6 Appendix

As an additional illustration of the mechanisms discussed in this paper, we present in this section an example from a second domain, that of software development. The overall goal of the example task is to create a new version of a software system, incorporating desired changes and additions. The primary agent initiating the task is the project leader, who is directing a programmer, *Dave.Hildum*, to effect the changes. POLYMER generates a partial plan network for this task, and executes it in conjunction with the relevant agents. There are three ordered subgoals generated for this task: *decide-on-changes* (the programmer must decide which particular changes to make), *make-changes* (the editing must be performed on the appropriate modules) and *have-consistent-system* (the entire software system must be updated so that changed modules are recompiled and the system is relinked).

The agent (Dave.Hildum) is instructed to perform the actions *think* and *edit* which are selected by POLYMER to achieve the first two subgoals. The actions are performed as anticipated. The planner attempts to achieve the third subgoal *have-consistent-system* by selecting the activity *update-software-system* to achieve it. Upon requesting verification from the user to perform the first primitive action in this activity expansion (*compile* the first changed file), the user denies verification and instead initiates a *unix-make* action. SPANDEX is invoked by POLYMER upon detection of this exception, and the exception classifier determines that an action mismatch has occurred, implying a possible attempt at an action substitution or an out-of-order action[4]. An *exception record* (see Figure 5 below) is created to summarize the exception, recording the action which generated the exception, and the type of exception which was detected. SPANDEX then invokes the exception analyst, which uses a heuristic *strategy.selector* to choose a strategy to generate an *explanation* for the exception.

[4]These implications are derived from relevant plausible inference rules, as described in section 3.1.

Unit-name: SUBSTITUTE.FOR.HIGHER.LEVEL.GOAL.01
Unit-comment: "Show that the unexpected action is a substitute for an in.progress.parent.node of the expected action."
Explanation-summary: "The unexpected action *unix-make-01* is sufficient since it has a goal unifying with the goal of the parent node *update-software-system-01*. "
Status: complete
Reasoning.method: substitute.for.higher.level.goal
In.progress.parent.activity: update-software-system-01
Pending.goal.achieved: updated(SPANDEX)
Unexpected.action.goal: updated(SPANDEX)
Hierarchy.level.difference: 2

Figure 6: Explanation for ACTION.MISMATCH.01

In this example, a single applicable plausible inference rule is retrieved, and SPANDEX constructs one plausible explanation, which is complete. (see Figure 6). In this particular case, the complete explanation states that since the goal of the unexpected action *(updated(*SPANDEX*))* unifies with the goal of a parent[5] of the expected action node (the goal of *update-software-system*, which is the parent node of the expected *compile* action, is also *updated(*SPANDEX*)*) the unexpected action may be a substitution for the more abstract parent node. Since there is only a single explanation in this example, there is no need for negotiation among affected agents to choose among potential explanations.

An amendment is next constructed for the explanation which specifies the changes that must be made to the current plan network and domain knowledge in order to restore consistency to the system. The implementation of this explanation involves replacing the wedge of the plan network subsumed by the more abstract parent node *(update-software-system-01)* with the unexpected action *(unix-make-01)*. As a side effect, the nodes in the expansion of *update-software-system-01* are deactivated from the planner's predictions (see Figure 7).

The changes specified by the implementation field of the amendment shown are performed, and planning and execution by POLYMER is resumed.

References

[1] Borgida, A., and K.E. Williamson, "Accommodating Exceptions in Databases, and Refining the Schema by Learning from Them," *Proceedings of the Very Large Data Base Conference*, 1985, pp. 72-81.

[2] Broverman, C.A., Croft, W.B. "Exception Handling During Plan Execution Moni-

[5]The term "parent" here is used to refer to the more abstract node from which an expansion now in place in the current network was derived.

Unit-name: REPLACE.PLAN.WEDGE.01
Unit-comment: "Replace a wedge of the plan subsumed by a single node by a new node."
Amendment-summary: "Replace the plan wedge subsumed by
update-software-system-01 with *unix-make-01*."
Implementation: (do
(replace-wedge update-software-system-01 unix-make-01)
(deactivate compile-file-01 compile-file-02 compile-file-03
link-system-01)

Figure 7: Amendment for SUBSTITUTE.FOR.HIGHER.LEVEL.GOAL.01

toring," *Proceedings of the Sixth National Conference on Artificial Intelligence*, July 1986, Seattle, WA., pp. 190-195.

[3] Broverman, C.A., Croft, W.B. "SPANDEX: An Approach to Exception Handling in an Interactive Planning System," COINS Technical Report No. 87-127, University of Massachusetts, Amherst, Mass. December 1987.

[4] Croft, W.B., Lefkowitz, L.S. "Task Support in an Office System," *ACM Transactions on Office Information Systems*, 2: 197-212; 1984.

[5] Croft, W.B., Lefkowitz, L.S. "Knowledge-Based Support of Cooperative Activities," *Proceedings of the Hawaii International Conference on System Sciences*, January 1988, pp. 312-318.

[6] Croft, W.B., Lefkowitz, L.S. "A Goal-Based Representation of Office Work," IFIP Conference on Office Knowledge. North Holland, W. Lamersdorf, ed., 1988.

[7] DeJong, G.F. "Generalizations Based on Explanations," *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, Vancouver, B.C., Canada, August 1981, pp. 67-70.

[8] DeJong, G., Mooney, R. "Explanation-based Learning: An Alternative View," *Machine Learning*, 1, 1986.

[9] DeJong, G. "An Approach to Learning From Observation," Machine Learning, Vol. II. Michalski, Carbonell, Mitchell, eds., Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1986, pp. 571-590.

[10] Fikes, R.E. "A Commitment-based Framework for Describing Informal Cooperative Work", *Cognitive Science*, 6: 331-347; 1982.

[11] Hayes, P.J. "A Representation for Robot Plans", *Proceedings IJCAI-75*, 181-188, 1975.

[12] Sacerdoti, E.D. *A Structure for Plans and Behavior*, Elsevier North-Holland, Inc., New York, NY, 1977.

[13] Sathi, A., Morton, T.E., Roth, S.F. "Callisto: An Intelligent Project Management System," *AI Magazine*, 7:5, Winter, 1986, pp. 34-52.

[14] Tate, A. "Generating Project Networks," *Proceedings IJCAI-77*, Boston, 888-893, 1977.

[15] Wilkins, D.E. "Recovering from Execution Errors in SIPE," SRI International Technical Report 346, 1985.

Appendix 6-E

A Survey of Recent Planning Research

Christopher Eliot

1 Introduction

Researchers have studied the problem of generating plans automatically for almost as long as the field of Artificial Intelligence has existed. It is somewhat unique because there has been more agreement among researchers and convergence of opinion than is normally found in Artificial Intelligence. This makes a study of previous research a more important prerequisite to research in this area than it is in some other areas of AI. It also invites a different kind of analysis of that research. Many AI research projects stand as isolated explorations of their domain, having no direct predecessors and no direct descendants. What draws them together is a common set of basic techniques and an approach to building systems. To describe these domains it is sufficient to give a description of each research project in the domain and let it go at that. Planning, on the other hand, invites an attempt to classify and compare different approaches. Most planners are clearly based upon earlier planners. Historical trends can be traced showing how various aspects of planning have evolved.

The general approach of this survey is to follow a historical progression within a theoretical analysis. We analyze planning into two representational issues, time and state, and the heuristic issue of control. These issues are discussed in separate sections. Within each of these sections the discussion generally follows a historical progression. The earliest research addressing each issue is given a reasonably complete description. Descriptions of later research include only those features which that research adds or takes away from the previous ideas. The reader should have a clear understanding of the current status of each of these issues after reading the relevant section.

In all cases the emphasis is to provide a meaningful explanation of the significance and purpose of the various features, less than to provide a full description of all of the details. By simplifying everything as much as possible I hope to give the reader a clear introduction to planning. The details that I am leaving out can be found in the original literature. Inevitably, there will be something left out of this survey that should have

been included. I apologize in advance for any such errors or omissions, which are, of course, solely my fault.

2 Definition

Naively, planning is the creation of plans. Plans are an abstract representation of something more concrete, designed to facilitate both planning and the actual realization of the concrete thing. For example, the plans for a building are a set of drawings made to scale so that an architect can think about what the final building will be like, and annotated with engineering data so that a contractor can actually build the structure.

While this definition is circular, and thus not truly adequate, some things follow from it directly. It will be necessary to manipulate, inspect and analyze plans. The effort spent planning must "pay" for itself by reducing the overall effort needed to realize the concrete thing. There must also be a desire for the concrete thing. From this desire for the concrete thing we may infer the existence of a set of goals which define it.

Presently, we mean to focus on a specific type of planning. However, we may occasionally have uses for a collection of examples of planning of the general type that we are now discussing.

Certainly architecture is a form of planning. The actual thing desired is a specific type of building. The plan is the set of blueprints. The goals which define it are such considerations as how much money can be spent, which materials can be obtained, and the uses and durability that will be required. Parts of the blueprint can be erased and redrawn easily so it is much cheaper to create a plan for a building than to create the building directly. Finally, the plan can be analyzed to determine, for example, if the building is structurally sound, so the final thing produced may be better as a result of planning than it would be if it were realized directly.

A simple grocery list is also a plan. In this case the concrete thing is to purchase a set of items, and the goal is to have them so they can be used. The plan can be analyzed, for example, by comparing the dinner menu with the grocery list and the pantry to determine that everything on the menu is either in the pantry or on the list in sufficient quantity. The menu itself is a plan for eating and the recipes are a plan for cooking.

Sometimes we say that we are "planning" a lecture. The concrete object is the actual delivery of the lecture. The goal is to inform the audience of various things and to be convincing about it. The plan has several components. There may be an outline of the talk or an actual text. Points to be covered and ignored must be chosen. Charts, diagrams and pictures may be generated and made into slides of some sort. Finally we may try to anticipate some of the questions that might be asked and generate plans to answer them. These plans have the form of if-then rules indicating that *if* this question is asked *then* I will give that answer. There will be more to say about this kind of planning in the section on reactive planners.

Inspection of these samples of plans, and others like them, reveals that many plans

consist of pictures or representations of things, while others correspond to procedures for doing something. For example, a recipe must indicate the order in which its various steps are to be done, but there is nothing that looks like a cake in a recipe. On the other hand, a blueprint for a building is clearly a picture of the building, but it does not indicate which parts of the building will be constructed first. The problem we will focus on, which is called the conjunctive planning problem, is the transformation between pictures or representations, which we will call designs, and a set of steps needed to instantiate a design. From now on we will call this set of steps a plan.

For example, there are many steps before a building is actually constructed. First an architect must transform the requirements for the building into a set of blueprints. Then an engineer must analyze the blueprints to determine (if possible) some order in which the parts of the building can be put together. This results in a choice of building steps and a timetable for doing them. Similarly, a menu must be transformed into a set of steps needed to cook the items on the menu. Although the menu defines a temporal sequence of courses, the process of preparing the food may completely mix up that temporal sequence. For example, a chilled dessert may be the first thing that has to be prepared even though it is the last thing on the menu.

A plan, then, is a set of actions which are to be executed in some temporal order to achieve some goal. Planning research, as we shall see, focuses on finding a workable set of actions and computing their effect on the world.

This is not the only view of planning. Some researchers argue directly against this view [1]. In their view the separation of planning from execution is a serious mistake. Planning cannot be thought of as producing a "program" that is "trivially" executed. The computational complexity of such an endeavor is too great and the real difficulties of carrying out a plan are difficult to investigate from this point of view.

In this survey we generally maintain the traditional view of planning. It is important to be conversant with the traditional view of planning when trying to understand the research, even that research which differs from the traditional view. Furthermore, this is the most highly developed view of planning and thus serves as the best foundation from which to proceed.

3 Early History of Planning Research

The STRIPS system is one of the earliest planners and also one of the most important. Many of the fundamental concepts of contemporary planning research were present in the STRIPS system. STRIPS was designed to control a robot that manipulated blocks in an artificial environment consisting of several connected rooms. It was asked to move blocks into new arrangements using primitive operators for moving the robot and pushing blocks. The STRIPS system would generate a complete plan for achieving a specified goal, which was then carried out by a separate system, called PLANEX.

Time was represented by a linear sequence of discrete points. The state of the world

at each point was represented by a set of sentences in the first order predicate calculus. An existing theorem prover, QA3 was used to prove goals and sub-goals. Crucial to the performance of STRIPS was the separation of the theorem proving control structure from the planning control structure. The planning control structure was means-ends analysis.

From an abstract point of view most planners are similar to STRIPS. Various elements have been improved but the overall relationships are the same. The input to the program is a goal and a data base of domain specific operators. The primary data structure and output of the program is a plan. The operation of the program is a heuristic search through a space of these plans, terminating when an adequate plan is found. The plan data structure consists of two parts, a temporal matrix and a set of state representations arrayed over the temporal matrix.

ABSTRIPS was identical to STRIPS in all aspects except its planning control structure, which used an augmented data base of operators to generate abstract plans. The abstract plans were used as skeleton plans from which to generate more detailed plans. The effect of this strategy was to check the overall coherence of a plan in terms of the most abstract operators before filling in the simpler details. It produces larger plans more efficiently than STRIPS, but depends upon the accuracy of additional domain knowledge.

Another fundamental system was HACKER. This system is now considered a planner even though it addresses planning issues in the guise of automatic programming. HACKER employs many different knowledge sources and was capable of a number of interesting forms of reasoning but the two ideas of most importance for planning were modifying plans (programs) to correct failures and the recognition of important anomolies that can occur during planning. In particular it was noticed that a planner could be led into a trap if the wrong subgoal is solved first. The importance of this was not immediately recognized but it is one of the central aspects of the conjunctive planning problem.

The NOAH system introduces the idea of a partially ordered procedural net, and hierarchical decomposition. The procedural net is an extension in the representational power of the temporal matrix. A more powerful representation introduces the possibility of more complex interactions among steps in a plan. NOAH defines a set of plan modification operators, called *critics* to resolve these interactions.

Each critic specialized in one class of possible interactions. It was a piece of code which inspected the plan to determine if its special interaction was present and, if so, then modified the plan to try to fix it. More recent planners no longer have distinct critics since the kinds of interactions that can occur have been better analyzed and general algorithms can be implemented that detect and correct all interactions.

Hierarchical decomposition is another refinement of the planning control structure. Like ABSTRIPS it operates by generating a succession of more detailed plans. However, there is a difference in which plans are considered more abstract than others. In ABSTRIPS certain conditions are considered more easily changed than others. Unachieved conditions appearing anywhere in the current partial plan are treated as subgoals. In ABSTRIPS the more difficult subgoals are always solved first while the less difficult subgoals are always ignored until later, regardless of the context in which they arise.

NOAH uses an opposite definition of which subgoals must be solved at each level of detail. It has no notion of which subgoals are more difficult, but relies entirely on the context in which the subgoals arise. At each iteration every unsatisfied subgoal is expanded, generally producing a subplan to achieve it. The subplan may include new subgoals that will be expanded in the next iteration. This implies that the (operational) definition of how abstract a subgoal is is the number of goal-subgoal links required to reach the original problem. In other words, the degree of abstraction is a measure of how much the subproblem contributes to solving the overall problem.

The foundations of planning research were capped off by NONLIN, developed at Edinburough by Austin Tate. It is fully described in [31]., In most respects NONLIN is very similar to NOAH. It benefits from a more careful analysis of the algorithms and problems. NOAH uses an unstructured set of global critics to resolve interactions in the procedural net. NONLIN considers all of the problems addressed by global critics to be protection violations and uses a simple algorithm to detect and resolve all of them. Additions to the plan library are made using a restricted declarative plan schema rather than with arbitrary pieces of code. Thus NONLIN marks a move away from highly heuristic planners back toward more algorithmic implementations.

NONLIN also introduces the idea of simple goal satisfaction. In some cases a subgoal does not have to be expanded into a subplan because it was already true earlier in the plan. It may be possible to add constraints to the plan (Before and After constraints on steps of the plan) so that the subgoal is satisfied. Achieving a subgoal this way is called *simple goal satisfaction*. No new steps are added to a plan by simple goal satisfaction so it helps produce efficient plans. Furthermore, no harmful interactions can be caused by simple goal satisfaction.

While it has no great theoretical significance, NONLIN was implemented so that it could backtrack if the plan it was working on could not be completed. Since NOAH would fail in these conditions this increases the number of problems that can be solved.

4 Classification of Planners

The bulk of this survey is an organized discussion of recent planning systems. The organization of this discussion follows the typical structure of these planners.

All planners must represent the relationship between different points or intervals of time, and must represent the state of the world at those points or intervals. The representation of time serves as a matrix or substrate upon which the different states of the world are placed. Design tasks are in many ways very similar to planning tasks, except that the states are arrayed over a spatial matrix rather than a temporal matrix. The power and generality of the temporal matrix has grown as planners have evolved. In early planners the temporal matrix is a simple linear sequence of points in time, corresponding to specific events. Later planners introduce partially ordered sets of points, iteration and conditional plans. Putting conditions into the plan matrix extends it so

that it is no longer simply a representation of temporal placement. The matrix actually represents several possible worlds.

A planner must also have some scheme for representing the states of the world. There has not been as much variation here as with the plan matrix. Most planners use some variety of first order predicate calculus to represent states of the world.

There are some planners which do not fit into this classification directly. Instead of addressing the representation of the plans these systems either propose new methods of generating plans, or ways to proceed under unusual constraints. Case based planning, reactive planning and meta planning are examples of the former while distributed planning is an example of the latter. Finally, some researchers are concerned with analyzing planners rather than immediately extending their capabilities.

5 Overview of Recent Research

It is useful to view NONLIN as marking a transition in planning research. By this time the basic planning problem defined by STRIPS was reasonably well understood. The basic techniques used to solve this problem, commonly called the classic planning problem, are embodied in the NONLIN system. Unfortunately the problem itself defines a limited view of the world and consequently its solution is of limited use. Research since then has been aimed at understanding and eliminating these limitations.

The problem with the classical approach to planning is that most of the complexity and behavior of the world cannot be described realistically. Even with this simplified representation the forms of plans that can be generated are still quite limited. Some of the most important aspects of the world that cannot be properly handled are multiple agents, independent processes and context dependent effects of actions. Some important limitations of the form of plans are the lack of conditional actions and iterative behavior.

Some simple information about the world is extraordinarily difficult to express using the predicate calculus common in planners. For example, a map is easy to describe geometrically but not easy to describe in predicate calculus. This suggests that extending planners to use more realistic representations of the world may cause the planning problem to become computationally intractible. In fact this is a major concern of recent research. Some results indicate that small extensions to classical planners make the planning problem np-hard [8], [7].

There are several different situations involving multiple agents. The agents may vary from being fully cooperative to completely uncooperative. Furthermore the ability to communicate and the reliability of the agents may vary. For example, a centrally controlled robot factory with many robots represents the simplest of these problems. All of the agents cooperate fully toward achieving the same goals. A more complex situation occurs when the robots are each responsible for making their own plans, with the central facility used for communication among the robots. This is called distributed planning. Since there is some limit to the bandwidth of any communication facility attempts must

be made to minimize or at least place an upper bound on requirements that each robot tell the others about its plans. In the most extreme case of distributed planning there is no communication facility, so each agent must determine the plans of other agents entirely by observation and extrapolation. It is still assumed that the agents want to cooperate and failure of any agent is to be avoided.

Planning is even more difficult if uncooperative agents must be taken into account. In situations where other agents merely have their own goals and are not concerned with those of the planner the planning problem becomes negotiation. For example, each robot in a factory may be given a specific task to complete while competing with other robots for general resources. Each robot is not specifically concerned with its effect on the ability of others to complete their tasks, but when a conflict arises the agents involved will exchange information and attempt to negotiate a mutually satisfactory resolution to the problem. For example two robots might both need to use a wrench and screwdriver to accomplish their tasks. If each robot picks up one tool then a deadlock can occur which must be avoided or resolved.

Finally planning can occur in situations where other agents are actively hostile. Game playing and warfare are examples of this situation.

Another important limitation of classical planners is being unable to represent actions whose effects cannot be precisely determined. Some actions have effects that are truly random, like rolling dice. The possible values and probabilities can be computed, but within those limits there is no way to predict the actual outcome when the dice are thrown. Other actions have context dependent effects. Placing a weight on the upper end of a see-saw will flip the see-saw unless there is a larger weight on the lower end. The effects of the action can be predicted, but to do so requires knowledge about the state of the world. This problem makes detailed planning very difficult because almost any action can have a variable effect if it is analyzed in sufficient detail.

6 Representation of Time

All planners depend upon some form of temporal matrix so the power of the temporal representation directly affects the power of the planner. In order to use a powerful temporal representation a planner must be able to determine what will be true when various parts of the plan are being executed and how modifications to the plan will affect this. Since a plan is being continuously modified during planning it is important to be able to update the temporal matrix quickly.

Most planners represent time as a set of discrete states. Some temporal logic is used to define the time when each state will occur during execution of the plan. What it means for a state to occur can vary, and will be discussed fully below. The transition between states represents actions that will be taken when the plan is executed.

Figure 1 illustrates the temporal matrix for the state based planners described below. It depicts a very simplified view of an incomplete plan that might be generated by these

systems. The figure shows how the planners would represent seven of the states involved in brewing coffee, using coffee beans that must be ground and brewed.

In state based planners the temporal matrix takes the form of a set of nodes, representing states and a logic defining the constraints among the nodes. In the earliest planners, STRIPS, ABSTRIPS and HACKER [30], the only constraints that could be used were Before and After. Furthermore all nodes in the plan had to be ordered with respect to every other node in the plan. Therefore the plan has the form of a simple linear sequence of actions [see Figure 1a]. We call planners of this sort *Linear Planners.* NOAH [24] was the first planner to relax this restriction. Plans in NOAH have the form of a partially ordered network of actions [see Figure 1b].

When two actions in the network are not constrained to be Before or After each other they can be performed in either order, provided all other constraints are satisfied.

The advantage of this is that it allows the planner to use a heuristic strategy called ***delayed commitment.*** The idea of delayed commitment while constructing a plan is to avoid making arbitrary choices that can cause expensive failures. The difficulty is that representing precisely the constraints on the plan that are known to be necessary requires the representation of very fine grained states of *partial knowledge* [16]. At some point the complexity of maintaining all of the partial knowledge becomes too expensive.

One way that a bad choice can lead to failure is that the planner may find it must impose a new constraint on the plan that conflicts with an old constraint. One way to recover from this is to backtrack to a point before the first constraint was imposed on the plan, and choose a different way to continue constructing the plan. Hopefully this will not lead to the same contradiction.

An alternative to this is to provide ways for the planner to repair a plan that contains incorrect choices. HACKER [30], [33], CHEF [15], [5],[6] and [17] all try to do this.

In DEVISER [34] the temporal logic also includes constraints on the duration and start time of actions. While there are many limitations of this representation it is powerful enough for the system to solve one of the most interesting examples in the planning literature. DEVISER plans activities for a simplified model of the two Voyager spacecrafts [34] (P.257). In addition to the actions that the spacecraft can perform, such as changing its orientation and turning on the cameras, there are scheduled events such as an Earth occultation (when the earth is hidden behind a planet or moon from the view of the spacecraft). This event is significant because the spacecraft cannot transmit data to Earth during an occultation.

Each node has a start time "window" and a duration. The window specifies the earliest, latest and ideal times for the node to start. These can be specified as actual times, e.g. 8:23:01, or using expressions involving functions and plan variables. When one activity precedes another these windows must be adjusted so that the first activity is finished before the second one begins. This may be impossible, in which case the planner must backtrack. It also may allow the planner to deduce that activities are ordered because the start time of one is after the finish time of another.

Figure 1c shows the same temporal structure as Figure 1b, except that the Earliest

Start Times (EST), Last Start Times (LST) and Durations are shown. For example, Boil Water is a goal node which may start from 0 - 100 minutes and requires 10 minutes to complete. Because the spacecraft modelled in the DEVISER example is capable of performing some actions in parallel, DEVISER produces plans that are unrealistic for a single agent to perform. Notice that the figure shows the start time for Grind Beans as 30 - 108. This assumes that the subplans for Get Grinder and Get Coffee Beans can be done in parallel. If DEVISER were modified to assume that all actions would be carried out by a single agent it would produce the plan shown in figure 1d. This is identical to figure 1c, except that the start times have been modified to reflect the fact that both Get Grinder and Get Coffee Beans must be carried out sequentially before Grind Beans can start.

The temporal matrix in DEVISER is strictly more detailed than that used in NONLIN. The planning problem is more difficult because DEVISER can detect more problems in a plan than NONLIN. Conversely a plan produced by DEVISER satisfies scheduling constraints as well as ordering constraints. In general more detailed representations make planning more difficult, but more accurate, while more abstract representations can make planning simpler by allowing delayed commitment. Notice that the ability to represent detail is distinct from the ability to represent abstraction. A planner may be restricted to reasoning about plans in great detail or very abstractly, or it may be able to reason at several levels of detail simultaneously.

James Allen has explored the problem of using more abstract representations of time in planning [4], [21]. The basic representation of time used by Allen is an interval. A large number of relations between intervals are defined allowing the construction of arbitrarily complex subsets of time. For example, it is possible to say that interval A ends before interval B starts, OR interval A starts after interval B ends. Because of this powerful temporal matrix a planner based upon Allen's logic is capable of representing a subspace of NONLIN's search space as a single plan. This can make planning more efficient because the planner will not have to backtrack as often. However, the temporal logic is so powerful that updating the temporal matrix when a new constraint is added can become more expensive than the backtracking implied by a simpler temporal logic.

Another set of proposals for extending the representational power of the temporal matrix is inspired by analogy to programming languages. The temporal matrix of a plan can be compared with the control structure of a program. This suggests that the basic control statements from programming languages, iteration and conditionalization, should be used to express plans. So far no one has described a working implementation that is capable of generating plans with these constructs but there has been discussion of them. Mark Drummond [9] presents a formalism called *plan nets* for representing iterative plans. He also discusses the generation of plans with sensory actions that allow the execution monitor to aquire information about the state of the world and choose ways to continue executing the plan based upon this.

Amy Lansky and Michael Georgeff [17],[13] have applied ideas from the study of concurrent programming to generating plans involving multiple agents. They are concerned

with finding adequate ways to specify the kinds of coordination that multiple agents require. Lansky uses the concept of group (not the mathematical concept) to specify the modularity of a plan. By giving the planner information about which elements of the world can affect each other it is hoped that groups of elements can be treated independently, leading to much greater efficiency when reasoning about complex problems.

Several researchers have proposed logical formalisms of time for use in planning [29],[9],[20]. It is extremely difficult to compare these formalisms with each other or with other approaches to planning. One major theoretical concern is to display rigorous deductive mechanisms to solve the major bugaboos that plague planning, such as the Frame problem or the Qualification problem. Another strain of theoretical research attempts to describe reasonably standard planners rigorously in order to derive theoretical descriptions of their limitations. David Chapman shows that planning is undecidable and that planning problems using extended plan representations are np-hard [7]. Dean [8] also states circumstances under which planning is np-complete.

These results lead both researchers to consider planners which are sound but not complete. A planning algorithm is sound if every plan that it produces is correct. It is not complete if there are solvable problems which it cannot solve. Some researchers believe that a major reformulation of the planning problem is necessary [2]. In their view planning and plan execution must be fully integrated in order to handle the complexity problems. A great deal of effort can be saved by waiting to see how the state of the world changes, instead of depending upon an expensive detailed prediction of the future. The hope is that by taking more of a wait-and-see attitude towards planning enough effort will be saved so that planning in substantially more complex situations is possible. Equally or more important to these researchers are problems with the execution of plans after they have been generated. They argue that a robot or other plan executor would seldom succeed in "blindly" applying a plan, since there will be random or unanticipated changes in the world that will invalidate a plan.

However, such a pessimistic attitude towards the traditional planning paradigm is not inevitable. Any general purpose planner must be "programmed" by making a domain description available. It is reasonable to compare this domain description to a programming language. If this comparison is accepted then the computational complexity of planning is of no concern. To do so would be analogous to criticizing FORTRAN because it is possible to write a non-terminating FORTRAN program. Instead it focuses attention on the content of the domain description and implies that the knowledge engineer responsible for the domain description must pay attention to efficiency issues. To require the domain description language to be unable to express computationally intractable plans is not practical. Instead we should consider how best to support a knowledge engineer using a potentially dangerous tool. Programming languages can express undesirable algorithms. Programming is made easier not by designing languages that are incapable of expressing anything wrong, but by using constructs that are less likely to lead to serious problems.

The problems with plan execution are very real, but the lesson to be drawn from this is not necessarily clear. It appears that "blind" execution of a plan is bound to fail and

plan execution may require more machinery than planning does. In fact, that is the whole point of planning. Plans and planning are useful precisely because they are simplified views of the future. They are simplified so that we can effectively reason about them. They are views of the future so that actions can be done at the right time and grouped for optimal effect.

A comparison with human planning may clarify these points. Certainly when people carry out plans they are capable of using the full resources of their cognitive capacities. However, they do form plans and in some cases stick to their plans despite apparent ways to do better. There are a number of reasons why this is so. Vacation plans are a good example. Taking a taxi from an airport to a hotel is a valid action at the planning level. The details of finding the taxi stand, waiting for the taxi, dealing with the driver etc., must be dealt with, but not while planning the vacation. For planning purposes a taxi ride goes from point to point, takes one hour and will cost $50. An important human ability is the capacity to notice when an approximation like this has been used in a critical situation, and to then plan more carefully.

Now, while a travel plan is very incomplete in many details, it contains elements that are not subject to reconsideration. By planning ahead it is possible to obtain tickets and reservations and do so using an optimal amount of money. However, having done so makes it difficult to change the plans "at runtime."

7 Representing States

Most planners represent time as a set of distinct states organized into some form of temporal matrix. We have discussed the various types of temporal matrix that have been used without describing the details of how the distinct states are represented. In fact most planners use a restricted form of first order predicate calculus. In fact most planners use *the same* restricted form of first order predicate calculus. However, there are some different representations that have been tried, and the ramifications of the state representation depend upon the temporal matrix that is used. Changing the temporal matrix can have dramatic effects upon the algorithms that must be used to properly implement the state representation.

Even though most planners use the same representation for facts it is instructive to discuss them in general terms before discussing specific representations. This makes clear how little depends upon the actual representation of facts and how much more depends upon the way they are organized.

The general idea behind most state representations is that a moment or period of time can be adequately described by a set of facts which are said to be true at that time. The temporal states of a planner always refer to the predicted state of the world during plan execution. In order to reason about a predicted state a planner must effectively compute the results of simulating the execution of those parts of the plan prior to the predicted state. The planner starts from an initial state and a goal state. A series of partial plans

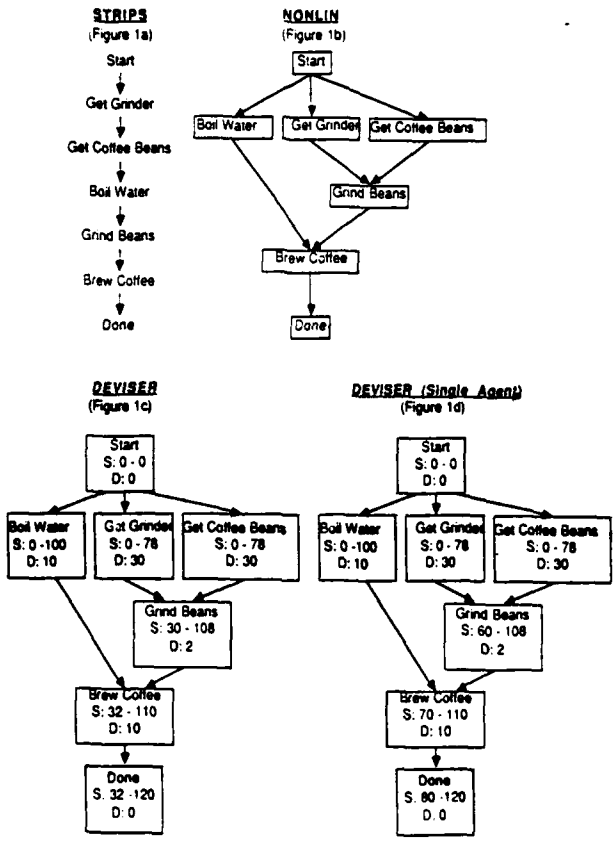


Figure 1: a, b, c, d

are generated until a complete plan is found that transforms the initial state into the goal state. Every partial plan must be simulated to determine if it transforms the initial state into the goal state.

It is important to have a simple conceptual model of this simulation, because the technical details and subtle pitfalls are extraordinary. At least three difficult considerations must be simultaneously addressed for any simulation technique to be adequate for planning. First, the particular domain must be described to the planner in the form of a knowledge base. There must be a tractable language for describing the constraints and operators in the domain that can be used by the simulation process. Second, the simulation must be able to operate efficiently even though the plan is frequently modified. Finally, it must produce reasonable and useful results when simulating an incomplete plan.

Simulating an incomplete plan requires manipulating partial states of knowledge and keeping track of the modality of facts. There are four important modalities (see Figure 2). In any state there may be facts known to be true, facts known to be false, facts required to be true and facts required to be false. The simulation process is generally invoked in response to the required facts and produces the known facts.

In order to be efficient a number of technical devices are used. Required facts can be structured according to *why* they are required and (if this is not the same thing) *what* they will be used for. Goal structures [31], protection intervals and dependency structures have been used for this purpose. Additional structure can be stored for known facts to provide fast modification to the plan when assumptions are retracted or superseded. This is called reason maintenance or truth maintenance. Known facts are not stored explicitly in every state. Instead assumptions are made about the persistence of facts and the temporal matrix is used as an inheritance structure. If a fact is true in one state of the plan and the assumptions about persistence are satisfied then it is considered to be true in following states. Only those facts which violate the assumption of persistence are stored explicitly. Consequently changing the temporal structure of a plan does not require expensive recomputations of the persistence of facts.

Reasoning about the persistence of facts requires planners to address the frame problem [26], [13]. The frame problem is to define when facts or assumptions will not be true or will not persist to be true. Two aspects of this problem often require separate treatment in planning. One aspect of the frame problem is that a planner will have to work with an incomplete description of the world. The second is that it must work with an incomplete understanding of the world. For example, the behavior of a meteorite is well understood but their presence or absence will generally not be included in a description of the world. On the other hand, there will always be some limit on the computation of consequences even of the facts that are known. A classic example of the frame problem is the potato in the tailpipe problem. A car will not start if there is a potato in the tailpipe. If a planner knows that there is a potato in the tailpipe it can predict that the car won't start, provided that it has sufficient understanding of the world of internal combustion engines (or special purpose rules). However, if the planner doesn't know about the potato then no amount of additional general knowledge will improve its prediction.

STRIPS makes very strong assumptions about persistence and provides a very weak temporal matrix. It is assumed that any fact will persist until the planner does something that makes it false. The temporal matrix is a linear sequence of states, so the simulation process is able to perform a simple forward sweep through the states to generate predictions of future states. The operator descriptions contain explicit lists of which facts will be deleted and added when the operator is executed. The transition between two states corresponds to the application of exactly one operator. At each stage in the forward sweep the facts in the delete list of the next operator are removed from the current state, and the facts in the add list are added to generate the next state.

Using a partially ordered set of states instead of a total order makes the same state representation much harder to use. The planner must be able to merge the add lists and delete lists of several states. Instead of a simple linear scan, something akin to the subset construction of a DFA from an NFA must be used [18]. In considering a particular state the simulation process must apply the add lists and delete lists to the imediately prior state. However, in a partial order there is a set of states which may or may not be immediately prior to whatever state is under consideration. An algorithm for handling

this is given in [31]. This is specified as an inference procedure in [8] and as a modal truth criterion in [7]. Dean provides an axiom from which it can be deduced that a true fact will be true at a later time, and another axiom which allows us to deduce that a fact is true throughout an interval [8](P.199). He gives both "weak" and "strong" versions of these axioms, called "projection" and "true throughout." Chapman gives an analgous criterion for modal truth and an equivalent non-deterministic procedure for making a fact be true [7](P.342). The full modal truth criterion is given below.

> Modal Truth Criterion. A proposition p is necessarily true in a situation s if two conditions hold: there is a situation t equal or necessarily previous to s in which p is necessarily asserted; and for every step C possibly before s and every proposition q possibly codesignating with p which C denies, there is a step W necessarily between C and s which asserts r, a proposition such that r and p codesignate whenever p and q codesignate. The criterion for possible truth is exactly analogous, with all the modalities switched (read "necessary" for "possible" and vice versa) [7](p.340).

This criterion and others like it are very general, but require substantial interpretation to be suitable as the basis for implementing a planner. Interpreting the phrase "possibly before X" is easy if the temporal matrix is linear. If the temporal matrix is a partial order this phase refers to all states constrained to be before X, and those states which are not ordered with respect to X. If the temporal matrix can represent conditional steps or iterative steps then this phrase is even harder to interpret. Similarly the codesignation of propositions is easy when propositions are represented as simple literals. Variables, connectives, quantifiers, deduction rules and sensory actions all present additional issues that make it harder to compute when two propositions codesignate. Deduction rules, in particular, make it difficult to even enumerate the propositions that are asserted in various states of the plan.

There are several ways to interpret a non-linear temporal matrix. One of the most basic questions is who will execute the plan. A plan can be generated without regard to who will execute it. In this case it describes the set of actions that must be performed, but does not account for limitations of the agents that will perform it. Most often it is assumed that one particular agent will execute the plan, and limitations of the agent are built into the description of the domain. Other planners assume that multiple agents will execute the plan [17]. One of the ramifications of the distinction between planning for multiple agents and planning for single agents was explained in the discussion of DEVISER above.

When plans are generated for multiple agents synchronization becomes a major concern. Many of the concerns of concurrent processing are relevant, such as deadlock detection and avoidance [22]. Various researchers [17], [13],[29] address these issues and import many ideas that have originated outside of AI to try to solve it.

8 Control

The control structure of planners is probably where the least variation occurs. Perhaps the only fundamental change that has been proposed since STRIPS is conceptual.

Even before STRIPS was implemented a planner based upon the resolution theorem prover had been built [14]. Essentially this planner operated by using an axiomatization of the domain in the situation calculus [19] to prove theorems asserting that there exists a plan to satisfy the goal under the initial conditions. This meant that there was no control of how the planner solved the problem, except the general strategy imposed by the underlying theorem prover. Consequently the planner was unable to solve non-trivial planning problems because it was too slow.

STRIPS introduced the idea of separating the heuristic control of generating the plan from the theorem proving process. Since the planning problem was found to be too complex to be solved as a general theorem proving problem the planner was structured so that a better heuristic control could be used. Previous representations used a situation variable to separate facts about different states. This adds complexity to the facts and means that the theorem prover had to deal with all of the facts about all of the states. Instead, STRIPS would present the theorem prover only with facts about a single state. Automatic deduction was limited to finding new facts within a state. The states were connected by operators with add and delete lists which provided the means of changing the set of true facts in different states. All facts not explicitly changed were assumed to remain as they were in the previous state. The facts are simpler, because the situation variable has been eliminated, and the theorem prover can operate more efficiently because it can focus on a smaller number of axioms at one time. Hence the overall planning process is much more efficient because of the separation of the heuristic control process from the theorem prover.

Since the theorem prover could not reason about operators or deduce the existence of new states, some other mechanism had to do so. Means-ends analysis, as implemented in Newell and Simon's General Problem Solver was used for this. In means-ends analysis each operator is indexed by the differences that it reduces. In each state with an open subgoal the difference from what is true in that state to the subgoal is used to retrieve an operator which is inserted into the plan before the state. Initially the only subgoal is the overall goal of the plan, which is a subgoal of the final state. Every operator which is inserted into the plan has preconditions which are subgoals for the state immediately before the operator is applied.

In STRIPS the difference measure used to index operators was derived from the failed attempt to prove the subgoal true. Resolution theorem proving operates by negating the conclusion and attempting to derive a contradiction. When this fails there will be some number of unresolved literals in the proof tree. (A literal is just a single predicate applied together with its variables or constants. For example, "NextTo(Robot, X)" could be a literal.) These were used by STRIPS as a difference measure. Operators were sometimes indexed under differences which the operator did not completely eliminate. For example

"flying" can be listed as an operator that reduces large geographic differences, even though some plan must be found to get to the airport of departure and from the destination airport to the final destination. Consequently the planning problem can be broken up into many smaller subproblems.

In some ways planners have progressed into simplified versions of STRIPS: the theorem proving capacity of most planners is much reduced or eliminated; means-ends analysis is still used, but generally without the ability to use partial matching; the state representation is basically the same; and operator descriptions are often very similar. The major feature of STRIPS that has become more complex in its descendants is the temporal matrix.

However, another view of planning has developed from Mark Stefik's Molgen [27],[28]. Instead of viewing planning as a search for a (generalized) sequence of operators that achieve a goal, planning can be viewed as successively constraining an initial plan until it completely and correctly achieves the goal. The temporal matrix and subgoals are all viewed as constraints. The subgoal constraints must somehow be satisfied for the plan to be complete. This view of planning is perfectly compatible with implementations such as NONLIN or DEVISER. Those planners can be viewed as heuristically choosing a constraint to satisfy and then replacing it with either a set of simpler constraints or an action (or both). There may be several different classes of constraints with different representations but abstractly they all serve similar functions.

Another way to think of the planning problem is heuristic search through a space of plans. Again this view is compatible with many of the planners that have actually been implemented. A heuristic search requires a starting point, or root node, which in this case is the original goal given in the planning problem, the ability to recognize a final node and a function to (heuristically) generate new nodes that may be closer to the target node. A node in the search space is a complete or partial plan. A final node is a complete and consistent plan. A plan is complete and consistent if all subgoals have been achieved and removed from the plan, the preconditions of all actions in the plan are satisfied and no other conflicts (such as protection violations) are present. In other words the search has found a plan that will solve the original problem. More accurately, we can only say that the planner has no knowledge of any way in which the plan may fail. Since any model of the world will in some ways be incomplete we must remain aware that incompletely modelled phenomena can lead a planner to generate incorrect plans.

The function which generates the next node in the heuristic search is more complicated. Every time the plan is modified in any way, a new plan is produced (at least conceptually). Strictly speaking, the whole plan should be copied, and the old plan should be saved as an alternate path through the search space. Actually, planners use various devices to avoid much of this copying. For the most part these optimizations are just standard programming tricks and special case improvements. The most advanced optimization along these lines is the use of intelligent backtracking methods, such as dependency directed backtracking. We will trace the development of this starting from Sussman's HACKER.

HACKER did not think of himself as a planner. Instead he generates programs to do some specified task. Since the tasks given as examples are often exactly the same as the goals given to planners it is possible to classify HACKER as a planner in retrospect, despite its development in the automatic programming domain. (In fact, planning and programming and code generation are very closely related.) HACKER would start by retrieving a program from its library, if possible, or else it could start from a null program. Since there would generally be some problem with the base program originally chosen it must be modified. In modifying the program it is likely that bugs will be introduced. HACKER then uses a set of ad hoc heuristics to fix the program. Primarily it is limited to replacing steps and reordering the steps. This debugging approach is used instead of backtracking.

SIPE uses backtracking in most cases but handles *resources* more efficiently [32](P.288). Resources represent objects which are temporarily used in execution of a plan. For example, a wrench or a coffee grinder can be viewed as a resource. A resource conflict arises when the plan calls for one resource to be used for two incompatible purposes at the same time. An example of this would be trying to use the same wrench to turn a bolt and hold a nut. When SIPE discovers a resource conflict it can try to bind the resource variable to a different object of the same class without having to backtrack. In this example it would have to find another tool to turn the wrench or hold the nut.

CHEF and Carbonell's systems are case based planners. Both systems start by trying to remember a similar planning problem that has been solved already. The previous *solution is then modified until it solves the current problem.* Both systems store an extensively annotated version of the previous solution and use the annotation to guide the process of transforming it into a new solution. CHEF indexes its previous solutions primarily by the anticipated *failures* that they avoid, while Carbonell's system indexes previous plans by *analogy* to the problems they solve.

When the *reason* for every choice made during planning is stored in a dependency structure it is possible to use dependency directed backtracking. In dependency directed backtracking every conclusion is stored with links to all of the facts which support it and all facts which are later derived from it. By annotating a plan (or any other representation of a reasoning process) with these extra links it is possible to implement backtracking much more efficiently. When a contradiction occurs the dependency structure is used to determine the minimal amount of changes that must be made to get back to a consistent plan. This is especially efficient if several large parts of a plan are logically independent of each other because a failure in one part will not affect other parts. Dependency directed backtracking as used in planning is discussed in [5], [6], [17] and [7].

9 Non-Traditional Planners

Certain planners do not fit into the classification scheme we have developed. Some of these are special purpose planners, that is, planners which are inherently useful for only one or

a few domains. We will ignore these completely. Others, we think, are not properly called "planners." Many of these systems are primarily some form of execution monitor. While this is an interesting and important problem in its own right, we consider it distinct from the topic of this paper. The two classes of non-traditional planners that we will discuss are *reactive planners* (defined below) and integrated planning systems.

There is currently considerable interest in the problem of integrating planning and execution. A strictly traditional view would consider these to be distinct phases with the plan serving as the data structure for communication between them. An opposite extreme is the position that every action must be based upon a full analysis of completely current sensory data. We believe that these extreme positions are both reasonable simplifying assumptions for research, but ultimately the real world is more complex. Generating, choosing and sticking to a plan provides a number of benefits. In many situations the demand for fast response does not allow sufficient time to fully analyze much of the sensory data that is available. In other situations there will be periods where no useful sensory data can be acquired and a plan must be followed as a form of dead reckoning. For example, in days gone by a sailing ship would be forced to simply continue its present course as best it could on days when the sun and stars were hidden by clouds. On the other hand it is also important to recognize when a plan should no longer be followed and to have strategies for proceeding in such circumstances.

Some tasks exist on a relatively large time scale and have strongly interacting subtasks. Constructing a building is one of these. These tasks demand tight organization and clear planning. Other tasks exist on a large time scale, but do not have strongly interacting subtasks. Constructing the many buildings in a city will serve as an example of this. These tasks can be well treated as a simple collection of smaller tasks. At another extreme we have tasks like balancing a broomstick or steering a car or playing a video game that require instant reaction to unpredictably changing events. We consider these to be dynamic control problems rather than planning problems. Our approach to these problems is to search for an appropriately ambiguous plan language in which the independent variables of dynamic control are not determined by the planner, but are left free for the use of lower level dynamic control mechanisms. These dynamic control mechanisms are properly biased by the planner in order to accomplish higher level goals. For example, a planner would not be asked to set the rudder of a sailing ship but only to choose a direction of motion. The difficult and unsolved problem is how to choose a plan language such that momentary and unstable decisions are left free but controlled by longer term, relatively stable decisions. In saying this we are advocating the position that part of the solution of the general planning problem is to characterize properly what is and what is not a planning problem.

Of course, other researchers draw the distinction differently. The study of reactive planners is a reflection of this. This unfortunate term is applied to planners which are intended to make their decisions in real time. Firby says "Systems that build or change their plans in response to the shifting situations at execution time are called "reactive planners" [12]. Since "reaction" and "planning" are essentially inconsistent ideas, the

term *reactive planner* is somewhat nonsensical. The improved term *participatory system* is used in [2] to describe a closely related type of system.

Firby describes a "reactive planner" in his AAAI paper [12]. His system operates by allowing a number of autonomous processes, called *Reactive Action Packages* (RAP) to simultaneously pursue their planning goals. Basically an RAP is a partially ordered network of goal and action nodes. A node can be executed when the earlier parts of the RAP have been and the node's preconditions are satisfied. A goal node is executed by activating and waiting for some other RAP which can achieve the goal. An action node is executed by invoking a (simulated) robot interface, which may provide sensory or other feedback. If an RAP cannot achieve its intended purpose it may fail, forcing the higher level RAP which invoked it to try another alternative.

In order to achieve real time execution he imposes some drastic constraints upon the RAPs. In particular the system is prohibited, as a matter of principle from reasoning about the future. He says that "This approach was chosen, not because an extreme approach would be a good planner by itself, but because reactive plan execution must occur at some level in every system." Unfortunately, this restriction is taken to mean that the system cannot reason about interactions, protection violations and other issues that are generally considered crucial. Because the system has such a trivial planner it cannot address many of the deeper questions of coordinating a planner with plan execution. In a domain where there are traps having serious undesirable consequences this system will fail badly where a more traditional system might succeed.

A related idea is *participatory systems* advocated in [2],[3]. This line of research is still evolving so it cannot be fully described. Their primary emphasis is on real time interaction with complex rapidly changing domains. To solve this problem they propose using mechanisms based rather directly on sensory input more than long range planning. Their ideas are illustrated in the Pengi system [3] which plays a simple video game. Essentially the system operates by building a description of the current state of the game, using this to choose a basic strategy and then also uses the description to implement the strategy. The idea of using the description to help implement the strategy allows their system to use a simpler symbolic description of the world. For example, the system may reason about concepts like the-bee-I-am-chasing, rather than having to locate and bind variables to BEE34. This is a form of appropriate ambiguity. If Pengi stops chasing BEE34 and starts chasing BEE35 a purely symbolic system would have to modify its state, but the-bee-I-am-chasing remains a sensible term.

10 Conclusion

One set of planning problems is now well understood. A number of remaining problems have been identified, such as dynamic domains with unpredictable events and multiple agents. However, most research treats planning in isolation—or at most, in combination with—some plan execution agent. In fact, planning should be integrated with other forms

of reasoning, such as diagnostic reasoning, deductive reasoning and reasoning with partial knowledge. Currently almost all planners demand perfect knowledge of the state of the world and the causal structure of their domain. In some domains unforeseen effects can be handled by failing and replanning, but this does not constitute reasoning about such effects.

Furthermore, almost all planners have implicitly assumed that the final result must be a plan in the canonical form of a set of actions. This assumption is probably undesirable. Most things which people take to be plans are not in this canonical form. For example, in planning a dinner party one might generate a guest list, a menu and a shopping list, none of which is a set of actions. By thinking of the flow of guests one may generate a sensible geometric arrangement for tables and chairs, hats and coats, refreshments and food. Ultimately, of course, an agent must plan and execute actions. Similarly a computer can only execute machine code. However, machine code is not the best language for reasoning about programs, and sets of actions may not be the best language for reasoning about plans.

11 Problems to Think About

One of the problems with planning research is that too much has been done with trivial examples. Where the examples are not trivial, they are often not comparable. Below are two problems which demand some additional planning capabilities, provide some measure of comparison between solutions, but promise not to be too intractible.

In the real world planning is an ongoing process. Things happen while plans are being carried out that must be responded to. Furthermore planning must take into account the cost of performing actions and weigh this against the value of the goals. I propose the collapsing block problem as a simple problem that forces these constraints to be considered.

12 The Collapsing Block Problem

The world consists of a table, blocks and a robot. The goal of the robot is to ensure that a tower of five stacked blocks exists as much of the time as possible. The robot can pick up a block and put it onto the table or on top of another block, requiring five units of time. After every 100 units of time a new block appears on the table which the robot can then use. Initially there are no blocks on the table. Unfortunately, there is also a chance that any block will collapse. During the first unit of time that a block exists there is no probability it will collapse. After each unit of time the probability that a block will collapse in the next unit of time increases by one tenth of one percent. Thus a block has a 10units of time after is was first created. When a block collapses it is destroyed and is eliminated from the world. Furthermore, all blocks stacked on top of a block that collapses are also destroyed.

Solutions to this problem can be precisely compared by computing the average number of complete five block towers that exist over a long period of time.

To solve this problem well requires making tradeoffs that will depend upon the situation. It is better to place older blocks on top of a stack because their collapse is more likely. On the other hand it takes time to disassemble a tower and place a new block at the bottom. Consequently the problem involves managing the resources so as to optimize the number of towers that are built before the blocks collapse. Furthermore the planner must take into account predictable scheduled events, (the arrival of a new block) and unpredictable problems of varying severity.

The Dinner Party Problem Solve the problem mentioned in the conclusion, namely to generate a guest list, menu, shopping list and furniture arrangement for a dinner party. Constraints apply at every step. For example, if there are any non computer scientists there must not be a majority of computer scientists or the others will be bored. Diabetics must have low-sugar diets. Turkey and cranberry must be served on Thanksgiving, and fresh foods are seasonal. Nothing should be on the shopping list if it is already on hand in sufficient quantities. Finally, from these lists generate a set of actions to prepare for and host the party. Demonstrate that the different forms of reasoning interact correctly. For example if some ingredient cannot be found and there is no substitution then the menu and theoretically the guest list might have to be revised.

The point of this problem is to think about different ways to express a plan. It should be possible to solve this problem with a series of planners, each having its own language for expressing the planning problem and the corresponding solution. Using special purpose planners this should not be hard. Can a single general purpose planner handle all of these problems? By actually building and analyzing such a set of special purpose planners it might be possible to better answer this question.

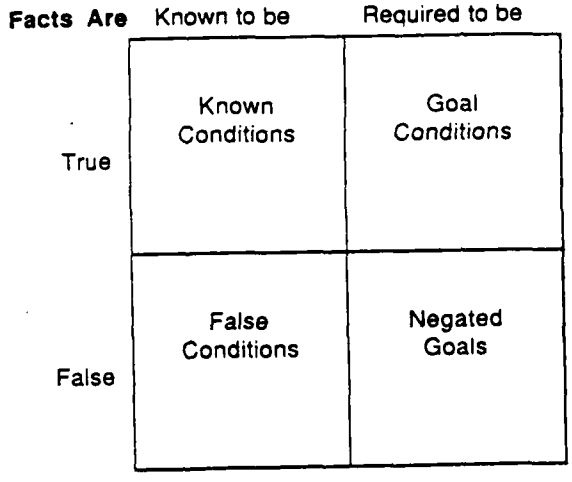


Figure 2

Part II: A Simple Example Planner

I have implemented a simple planner in order to provide a more concrete example than I have yet given. The program is based upon both NONLIN and SIPE but it is not a precise copy of either. The code is mostly written in Common Lisp, but the LOOP macro and Flavor System have been used.

The planner itself is domain independent. A domain description, called a plan library, must be given to the planner in order for it to function. The plan library consists of an arbitrary number of plan-schemas. Each plan-schema has a name, a list of local variables a list of effects and the body of the plan-schema. Any set of objects may be substituted for its variables to produce a valid plan fragment. The interpretation of this plan fragment is that "executing" the plan-body will result in a state where the effects are true. (No program has actually been written to "execute" these plans.) The plan body is a specialized constraint expression, consisting primarily of subgoal conditions and "actions."

From this plan library the planner can generate plans to achieve a goal. The plans generated are a set of actions structured into a network which defines constraints on the order in which the actions must be executed. The planner uses chronological backtracking to ensure that a valid plan is eventually generated, if possible. It must be emphasized that the definition of a valid plan is completely given by the plan library. One of the great frustrations of planning is to find a valid model of actions using the constraint language that is available. It is almost accurate to say that the entire problem of general purpose planning comes down to finding a sufficiently expressive constraint language with which to define a wide class of plan schemas, but which is also simple enough that an efficient planner can interpret it.

As an example I have implemented a plan library for the domain of brewing coffee. The primitive operators in this domain are actions like Buying something at a store, grinding coffee beans, boiling water and brewing coffee. These are related by their preconditions and effects. For example, to Buy something you must be at a store that sells it, and you must have money to pay for it. In order to Grind coffee beans you must have coffee beans and a grinder. Many details have been ignored, such as spoons and measures, cups and pots and the amounts of coffee, money and water. There also is no knowledge of how long it takes to do something, nor can any changes occur except under direct control of the planner.

It might seem that these simplifying assumptions are harmless enough so that the plans produced will be reasonable, if lacking details. However, this is not always the case. A number of obvious problems cannot be solved because of these simplifications. For example we must model money in one of two ways. Either Buying has an effect that says we don't have money or it has no such effect. If Buying means we don't have money then the planner must go to the Bank between every transaction. But without such an effect there is no conception of running out of cash either.

Another representational problem derives from the inability to model autonomous

processes. In particular the planner sees no reason why it shouldn't boil the water for the coffee before it starts to buy the coffee beans. Clearly (to us) this doesn't make sense. Either the water will be cold when the planner returns because an autonomous process has occured (cooling) or else the house will burn down. Certainly the planner is incomplete while it generates anomolies like this.

A third problem with the planner is in its representation of locations. Originally I modelled locations simply as names of places that the planner could "Be-At." The problem that arises is that the planner cannot determine where the planner no longer located is after it moves. Close inspection of the plan will reveal that it depends upon the planner being in two places at once. Instead, the movement plan schema might have as an effect that the planner is not at the location where it started from. This would remove the fact **(Be-At x)** from the later states, where **x** is the location the planner was at. However, this creates an undesirable restriction. By requiring the original location to be bound to a specific variable we are implicitly stating that the action moves the planner from one particular location to one other particular location.

At first this may not seem like a very important limitation. After all, it is hard to imagine how a person could travel from two starting points to one destination point, without having suffered a very unfortunate accident. It is not as harmless as it appears. An action that says "Go from location X to location Y" has the property that the planner knows the same amount about its location after the action is performed as before. An action that says "Go to X from wherever you are" has the property that the planner knows more about its location after the action is performed. When the planner is trying to predict the effects of this action it will actually reduce the ambiguity of its predictions. Admittedly some other process (the plan execution phase) must supply the information that the planner has apparently gained for free. But this is a fair division of labor, since the plan execution phase is not primarily responsible for the overall coherence of the plan.

One solution of this problem is to modify the way locations are represented. Instead of using the form **(Be-At***agent location***)** the modified planner expresses the location of the agent as the value of a function. This makes it possible to define a plan schema for travelling with an effect of the form:

(= (loc self) *destination*

Since **(loc self)** is an expression designating some unique object it is possible for the planner to deduce that **(loc self)** is the **destination** alone. For example, if the schema is instantiated with **destination** being **Bank** then the planner can deduce that **(loc self)** is not **Home**. Consequently it can determine that the precondition of the Grind plan is no longer satisfied and must be reachieved.

Appendix - Sample Plan Libraries

Plans are defined using the *Defplan* special form. The syntax of Defplan is:

```
(DEFPLAN  name variables
 (WARANTEE  sentences)
 (DECOMPOSITION  body))
```

The initial plan library is reproduced here.

```
(defplan Buy (x s actor)
  (warantee (have x actor) (not (have money actor)))
  (decomposition
    (protecting (shuffle (achieve (have money actor))
                                        (achieve (be-at s actor))
                                        (achieve (sells s x)))
                (perform (purchase-act x actor)))))

(defplan boil (actor)
  (warantee (have hot-water actor))
  (decomposition
    (protecting (achieve (be-at home actor))
                (perform (boil-act water actor)))))

(defplan make-coffee (actor)
  (warantee (have Coffee actor))
  (decomposition
    (protecting (shuffle (achieve (have hot-water actor))
                                         (achieve (have ground-coffee
actor)))
                (perform (filter-act hot-water ground-coffee actor)))))

(defplan get-money (actor)
  (warantee (have money actor))
  (decomposition
    (protecting (achieve (be-at Bank actor))
                (perform (withdraw-act actor)))))

(defplan Caffeinate (actor)
  (warantee (status alert actor))
  (decomposition
    (protecting (achieve (have coffee actor))
                (perform (drink-act coffee actor)))))

(defplan travel-by-foot (from to actor)
  (warantee (be-at to actor))
  (decomposition (protecting (achieve (be-at from self))
                (perform (walk-act from to actor)))))
```

When the planner is augmented to implement functional expressions in the constraint language it is possible to fix the problem with locations. The augmented plan library is shown here.

```
(defplan Buy (x s actor)
  (warantee (have x actor) (not (have money actor)))
  (decomposition
    (protecting (shuffle (achieve (have money actor))
                                        (achieve (= (loc actor) s))
                                       (achieve (sells s x)))
                (perform (purchase-act x actor)))))

(defplan boil (actor)
  (warantee (have hot-water actor))
  (decomposition
    (protecting (achieve (= (loc actor) home))
                (perform (boil-act water actor)))))

(defplan make-coffee (actor)
  (warantee (have Coffee actor))
  (decomposition
    (protecting (shuffle (achieve (have hot-water actor))
                                        (achieve (have ground-coffee
actor))
                                        (achieve (= (loc actor) home)))
                (perform (filter-act hot-water ground-coffee actor)))))

(defplan get-money (actor)
  (warantee (have money actor))
  (decomposition
    (protecting (achieve (= (loc actor) Bank))
                (perform (withdraw-act actor)))))

(defplan Caffeinate (actor)
  (warantee (status alert actor))
  (decomposition
    (protecting (achieve (have coffee actor))
                (perform (drink-act coffee actor)))))

(defplan grind (actor)
  (warantee (have ground-coffee actor))
  (decomposition
    (protecting (shuffle (achieve (have grinder actor))
                                        (achieve (have coffee-beans
actor))
                                        (achieve (= (loc actor) home)))
                (perform (grind-act actor)))))

(defplan travel-by-foot (to actor)
```

```
(warantee (= (loc actor) to))
(decomposition (perform (walk-act to actor))))
```

References

[1] Agre, P. E., and Chapman, D., "What are plans for?" *DARPA Planning Workshop*, October 1987.

[2] Agre, P.E., and Chapman, D., "From Reaction to Participation," *DARPA Planning Workshop*, October 1987.

[3] Agre, P.E., and Chapman, D., "Pengi: An Implementation of a Theory of Activity," *Proceedings of AAAI-87*, Morgan-Kaufmann, Los Altos, California, pp. 268-272 (1987).

[4] Allen, J. F., & Koomen, J. A., "Planning Using a Temporal World Model," *Proceedings of IJCAI-83*, Karlsruhe, West Germany, pp 741-747 (1983).

[5] Carbonell, J. G., "Derivational Analogy and Its Role in Problem Solving," *Proceedings of AAAI-83*, Washington D.C., pp. 64-69 (1983).

[6] "Learning by Analogy: Formulating and Generalizing Plans from Past Experience," In Michalski, J. G. Carbonell, and T. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, Tioga Publishing, California,, Chapter 5 (1983).

[7] Chapman, D., "Planning for Conjunctive Goals," *Artificial Intelligence*, Vol. 32, pp. 333-377 (1987).

[8] Dean, T., and Boddy, M., "Incremental Causal Reasoning," *Proceedings of AAAI-87*, Morgan-Kaufmann, Los Altos, California, pp.196-201 (1987).

[9] Drummond, M. E., "Refining and Extending the Procedural Net," *Proceedings of IJCAI-85*, pp. 1010-1012 (1985).

[10] Fikes, R.E., and Nilsson, N. J., "STRIPS: A new Approach to the Application of Theorem Proving to Problem Solving," *Artificial Intelligence*, Vol. 2, Nos. 3/4, pp. 189-208 (1971).

[11] Fikes, R. E., Hart, P. E., and Nilsson, N. J., "Learning and Executing Generalized Robot Plans," *Artificial Intelligence*, Vol. 3, pp. 251-288 (1972).

[12] Firby, R. J., "An Investigation into Reactive Planning in Complex Domains," *Proceedings of AAAI-87*, Morgan-Kaufmann, Los Altos, California, pp. 202-206 (1987).

[13] Georgeff, M. P., "Actions, Processes, and Causality" in *Reasoning About Actions and Plans, Proceedings of the 1986 Workshop* at Timberline, Oregon, M. P. Georgeff and A.L. Lansky (editors), Morgan Kaufmann Publishers, Los Altos, California, pp. 99-121 (1987).

[14] Green, C. C., "Application of Theorem Proving to Problem Solving," *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 219-239 Washington, D.C. May (1969).

[15] Hammond, K. J., "Learning to Anticipate and Avoid Planning Problems through the Explanation of Failures," *Proceedings of AAAI-86*, Philadelphia, PA, pp. 556-560 (1986).

[16] Kuipers, B. J., "Representing Knowledge of Large-Scale Space," Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Technical Report AI-TR-418 (1978).

[17] Lansky, A, "A Representation of Parallel Activity Based on Events, Structure, and Causality" in *Reasoning About Actions and Plans, Proceedings of the 1986 Workshop* at Timberline, Oregon, M. P. Georgeff and A.L. Lansky (editors), Morgan Kaufmann Publishers, Los Altos, California, pp. 123-159 (1987).

[18] Lewis, H. R., and Papadimitriou, C. H., *Elements of the Theory of Computation*, Prentiss-Hall (1981).

[19] McCarthy, J., and Hayes, P., "Some Philosophical Problems from the Standpoint of Artificial Intelligence," Stanford AI Project Memo No. AI-73 (1968).

[20] McDermott, D. V., "A Temporal Logic for Reasoning about Processes and Plans," *Cognitive Science*, No. 6, pp 101-155 (1982).

[21] Palavin, R. N., "A Model for Concurrent Actions Having Temporal Extent," *Proceedings of AAAI-87*, Morgan-Kaufmann, Los Altos, California, pp. 246-250 (1987).

[22] Peterson, J. L., and Silberschatz, A., *Operating System Concepts, Second Edition*, Addison-Wesley, pp. 307-401 (1985).

[23] Sacerdoti, E. D., "Planning in a Hierarchy of Abstraction Spaces," *Artificial Intelligence*, Vol. 5, pp. 115-135 (1974).

[24] Sacerdoti, E. D., "A Structure for Plans and Behavior," Technical Note 109, Artificial Intelligence Center, SRI International, Menlo Park, California (1975).

[25] Sacerdoti, E. D., "Problem Solving Tactics," *Proceedings of IJCAI-79*, Tokyo, Japan, pp. 1077-1085 (1979).

[26] Shoham, Y., "What is the Frame Problem" in *Reasoning About Actions and Plans, Proceedings of the 1986 Workshop* at Timberline, Oregon, M. P. Georgeff and A.L. Lansky (editors), Morgan Kaufmann Publishers, Los Altos, California, pp. 83-98 (1987).

[27] Stefik, M., "Planning with Constraints (MOLGEN: Part 1)," *Artificial Intelligence*, Vol. 16, pp. 111-139 (1981).

[28] Stefik, M., "Planning with Constraints (MOLGEN: Part 2)," *Artificial Intelligence*, Vol. 16, pp. 141-170 (1981).

[29] Stuart, C. J., "Branching Regular Expressions and Multi-Agent Plans" in *Reasoning About Actions and Plans, Proceedings of the 1986 Workshop* at Timberline, Oregon, M. P. Georgeff and A.L. Lansky (editors), Morgan Kaufmann Publishers, Los Altos, California, pp. 161-187 (1987).

[30] Sussman, G. A., "A Computational Model of Skill Acquisition," MIT AI Lab Memo No. AI-TR-297, (1973).

[31] Tate, A., "Project Planning Using a Hierarchic Non-Linear Planner," Research Report No. 25, Department of Artificial Intelligence, University of Edinburgh (1976).

[32] Wilkins, D. E., "Domain Independent Planning: Representation and Plan Generation," *Artificial Intelligence*, Vol. 22, pp. 269-301 (1984).

[33] Wilkins, D. E., "Recovering From Execution Errors in SIPE," Technical Note No. 346, Artificial Intelligence Center, Computer Science and Technology Division, SRI International, Menlo Park, California (1985).

[34] Vere, S., "Planning in Time: Windows and Durations for Activities and Goals," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 5, No. #3, pp. 246-267 (1983).

Appendix 6-F

An Ada Restructuring Assistant

Philip Johnson*
David Hildum*
Alan Kaplan*
Carolyn Kay°
Jack Wileden*

*Department of Computer and Information Science
°Department of Electrical and Computer Engineering
University of Massachusetts
Amherst, Massachusetts 01003

Abstract

Maintaining an appropriate structure for an evolving Ada software system is time-consuming and error-prone. This paper describes a new approach to automated support for restructuring Ada software based upon the artificial intelligence technique of heuristic search. We contrast our approach with other software restructuring systems, report our experiences with a small scale restructuring problem, and discuss the implications of this work for future research.

This work is supported by: the Air Force Systems Command, Rome Air Development Center, Griffiss Air Force Base, New York 13441-5700; the Air Force Office of Scientific Research, Bolling Air Force Base, District of Columbia 20332, under contract F30602-85-C-0008; the National Science Foundation grant CCR 87-04478 with cooperation from the Defense Advanced Research Projects Agency (ARPA Order No. 6104); and by the Office of Naval Research under Contract #N00014-79-C-0439, and under a University Research Initiative Grant, Contract #N00014-86-K-0764.

1 Introduction

Designing modifiable, maintainable, and reusable Ada software is a difficult process. Selecting an appropriate structure for a system and implementing it with the Ada package construct is fundamental to achieving these properties. Unfortunately, the original structure designed for an Ada software system often becomes increasingly inappropriate over the course of its lifecycle. This can be due to errors during the requirements, specification, design, or implementation phases, or to changes in requirements and/or specifications during system maintenance.

Correction of inappropriate structure can involve changing the package membership of *program objects* like procedures, functions, variables, and data types, as well as the creation of new packages or deletion of existing ones. This restructuring process is time consuming and difficult for several reasons. First, it is not always obvious how to remodularize the system—there may be several different "restructuring directions" which show promise, or the developer may not be aware of some restructuring alternatives. Second, it is difficult for developers to understand and resolve package interdependencies "in their head," which can result in several rounds of compilation while the compiler uncovers them. Third, selecting a new structure involves both the compiler-aided resolution of package interdependencies for *each* alternative, as well as their evaluation according to some criteria. Finally, in many cases, the structural requirements for a system may change not once but several times over the course of its lifetime as its context of use changes and functionality evolves.

Faced with the difficulty of restructuring an Ada system, and the prospect of eventual restructuring of the new structure, many developers opt for a "minimal impact" strategy of making the least amount of structural revision necessary to implement any modification. This provides only short-term benefits, with the long term cost being structure that does not reflect a natural decomposition of the problem, subtle and unintuitive structural dependencies, and tricky pieces of code. Such systems are difficult to understand, difficult to debug, and difficult to extend.

We have implemented a prototype of an Ada restructuring assistant which provides aid for the development, analysis, and generation of new structures for Ada software systems. Our assistant examines an Ada software system, creates a variety of representations of the system structure, allows the user to build and analyze new structures, and generates the restructured Ada source code. Most importantly, our assistant does not require the developer to completely specify new system structures. Instead, the developer can specify a "skeletal structure" in terms of the packages in the system and those program objects with an intrinsic package membership. The assistant assigns each of the remaining "free" program objects to packages according to heuristics for software structuring. In our prototype, we have designed a set of heuristics which loosely embody one "philosophy of system structuring." This philosophy attempts to generate systems consisting of abstract

data types with high cohesion, low coupling, and information hiding[1]. This "automated restructuring process" uses the artificial intelligence technique of heuristic search through the space of possible system structures. Our assistant finds *all* potential restructurings of the system possible given the initial skeletal structure and the constraints on potential system structures as expressed through heuristics. This allows developers to restructure an Ada system by specifying only those structural elements they are sure of, leaving the assistant to complete the restructuring process and provide a set of new structures to choose from. By reducing the overhead involved with manual generation and evaluation of new Ada system structures, as well as by providing a mechanism for heuristic "discovery" of new structures, we believe that our prototype shows promise in alleviating a significant bottleneck in the development and maintenance of Ada programs[2].

2 Related Work

Software "restructuring" most commonly refers to systems like RECODER [3], which inject structured programming constructs into unstructured code. These systems and other related efforts are reviewed in [1]. Like the assistant, these programs take the source code of a program and produce a reorganized program with equivalent semantics. The approach differs in the granularity of structure considered and the corresponding form of support. Rather than providing support for representing and reorganizing the modular structure of a system, they concentrate on much more local structural issues, such as the replacement of GOTOs by iteration and conditional constructs.

The CAPO system [5] addresses structure at the same level of granularity as the Ada Restructuring Assistant. CAPO supports the implementation phase of development by producing a system modularization from a design-level formalism similar to PSL [8]. The modularization process is guided by metrics assessing coupling, cohesion, reference distribution, information distribution, and transport volume. A very similar system to CAPO is KDA [7]. KDA also produces a single modularization from a design using coupling and cohesion metrics. The principle difference between CAPO and KDA is their control structure: KDA is modeled after the blackboard model of control, while CAPO uses clustering techniques like the median method of power or centroid clustering method. Both systems suffer from limitations which we became aware of through our work on a similar system, MIDWIFE [4], and which motivated some of the design decisions in the Ada Restructuring Assistant.

A primary problem with CAPO, KDA, and MIDWIFE is that they return a single modularization as "the" answer. Alternative modularizations require adjusting the

[1]Other structuring philosophies, such as an Ada-style "object orientation" [2], are possible with alternative formulations of the heuristics.

[2]The current version of our prototype fully implements all of the structural manipulation and representation features of the assistant. At this time, the prototype does not automate the parsing of the original Ada program or the generation of the code corresponding to new system structures. We expect the addition of these subsystems to be straightforward, however.

weightings assigned to the metrics or substitution of a different clustering algorithm. The systems force the developer to be either convinced that the modularization method is robust enough to find the best structure, or else simply content that the provided structure is "good enough." The manipulation of the weightings or clustering algorithm ameliorates the situation somewhat, but still requires a great deal of faith in the structural rating mechanism.

Unfortunately, this faith may be unwarranted, since the logic behind the structural rating mechanisms is suspect. An undisputed tenet of software engineering research is that good modularizations are generally characterized by low coupling, high cohesion, etc. It is a giant step, however, to conclude from this a strong form of the converse: not only that modularizations satisfying low coupling and high cohesion are good ones, but that minimizing coupling and maximizing cohesion yields an optimal structure. The general problem is that metrics like coupling and cohesion ignore the semantics of the application domain. It is quite possible to create modules with very low coupling and high cohesion which are not good system structures because they lump together semantically unrelated code.

The Ada Restructuring Assistant addresses both of these problems. Rather than return a single modularization, it returns every modularization satisfying the criteria specified by the developer through the heuristics. In addition, skeletal structures allow the developer to establish semantically related code groupings and thus incorporate crucial domain-level knowledge into the structuring process. Finally, we believe that concepts like "abstract data type" or an Ada-style formulation of "object orientation" [2] are more appropriate metrics for the problem of software structuring, in fact subsuming metrics like coupling or cohesion.

3 Using heuristic search to find new structures

Our assistant formulates new system structures by taking the skeletal system structure generated by the developer and assigning each of the free program objects to packages. Conceptually, this process can be thought of as generating a tree of system structures, where the root node is the original skeletal structure specified by the developer and each node's children "fill out" the partial structure of their parent by the assignment of some or all of the remaining free program objects to packages. If this tree were completely generated, the "leaf nodes" would be all of the solution states where every free program object in the original skeletal structure had been assigned to a package. It should be clear that for any system and free program object set of realistic size, a brute force approach to generating this tree is computationally intractable. Our assistant overcomes this problem by the use of three kinds of heuristics to control the search process. These heuristics attempt to prevent the tree generation process from exploring bad paths—those which result in clearly unacceptable system structures.

3.1 Assignment heuristics

Assignment heuristics generate the children of a partial system structure. These heuristics express "rules of thumb" about acceptable ways to structure a system. The dozen assignment heuristics in the current version of the prototype fall into the following three general categories:

- *Generate program object assignments to "hide" type implementations.* For example, this is accomplished for types by either assigning a type to the package where its implementing type is found or by assigning a type to the package where the types that use it are found. Successful application of this heuristic leads to the type becoming "private."
- *Generate assignments to hide procedures, functions, and types within a package.* This is accomplished by assigning a program object to a package where all program objects that reference it are found.
- *Generate assignments to collect related data and procedural program objects together in the same package.* This is accomplished by assigning data objects (such as types and variables) to packages where their manipulating procedural objects (such as procedures and functions) are found, and vice-versa.

3.2 Recommendation heuristics

Recommendation heuristics control the way the tree of system structures is expanded. Recommendation heuristics decide which of the current partial system structure nodes to expand next. In our prototype, the four recommendation heuristics attempt to choose the leaf node exhibiting the most "promise." The manner in which these heuristics are used is an important distinguishing characteristic of the Ada Restructuring Assistant from systems like CAPO and KDA. In these latter systems, the accuracy of their "recommendation heuristic" is crucial since it determines the single solution provided by these systems. In our Assistant, however, the recommendation heuristic simply helps the Assistant to find better quality solutions earlier in the problem solving process.

The promise is currently evaluated using these features of the partial structure:

- The closeness of the partial modularization to a solution (i.e., the number of remaining free objects).
- The number of hidden and invisible objects in the system. (The greater this number, the more promise.)
- The number of visible objects in the system. (The lower this number, the greater the promise.)
- The number of circular dependencies among packages. (The lower this number, the greater the promise.)

3.3 Pruning heuristics

Pruning heuristics find partial structures that are so unpromising that they should not be considered for further expansion. For example, partial structures that exceed a threshold value of circular dependencies among packages are pruned from further consideration for expansion.

3.4 Problem solving strategy

The assistant incorporates these heuristics into the following problem solving strategy:

1. An Ada program is loaded into the assistant.
2. The user generates the skeletal system structure through any or all of the following operations:
   (a) Assigning objects to different packages.
   (b) Deassigning objects from packages (making them free).
   (c) Creation of new packages.
   (d) Deletion of existing packages.
3. The assistant generates new structures by repeating the following loop until all structures are found or the user decides to stop.
   (a) Generate the children (new structures) of the chosen structure by invoking the assignment heuristics.
   (b) Remove unacceptable structures from further consideration by invoking the pruning heuristics.
   (c) Remove solution states (no remaining free objects) from further consideration.
   (d) Invoke the recommendation heuristics on the remaining partial structures.
4. Solution states (complete restructurings) can be displayed and evaluated by the user.
5. The user directs the assistant to generate any of the Ada programs corresponding to these new structures. (This feature is not implemented in the current version of the prototype.)

4 The user interface

The Ada Restructuring Assistant provides window-based interfaces in support of each phase of problem-solving. Figures 1, 2, and 3 illustrate the display of an initial, skeletal,

and solution system structure. The contents of each package are divided into three categories: *operations* such as functions and procedures, *types*, and *variables*[3]. The legend in the bottom right corner describes how the font used for each object's name indicates visibility information. All program object names are mouse-sensitive and can be directed to display further information about their corresponding program objects. The menu on the left hand side of the screen lists available operations. Those operations not available in the current context are italicized.

Figure 4 illustrates one alternative representation of a system—a graph of the calling hierarchy. Other alternative representations illustrate package dependencies; reference, change, and representation manipulation of objects by procedures; and variable-type relationships.

The assistant displays the progress of problem solving through a dynamically-updated graph of the search tree generated. An example of this window is shown in Figure 5. In addition to the graph, a scrollable window describes the progress of problem solving textually.

5 Results

Our initial experiences with the Ada Restructuring Assistant are limited but encouraging. We selected for experimentation a modestly-sized (120 program objects) Ada program developed to explore alternative machine learning strategies for the Blackjack card game. We chose this program because the initial structure contained a single package "Blackjack," which blended together the data structures and operations used to implement the card deck as well as those specific to the game of Blackjack. Clearly, separation of the card deck program objects into their own package would enhance the reusability and extensibility of this system. We were interested to see if our heuristics for expressing the philosophy of abstract data types and information hiding would be sufficient to allow the Assistant to accomplish this restructuring automatically.

We began by removing from the "Blackjack" package those objects intrinsic to our new 'deck of cards' abstract data type: the 'Card' and 'Ace_Value' data types, the constants 'Ten,' 'Jack,' and 'King', and the operation 'Shuffle_Deck'. Next, a small number of objects specific to the game of Blackjack were left in the "Blackjack" package: the data types 'Person' and 'Hitorstand,' and the constants 'Twenty_One,' 'Dealer_Stick,' and 'Eleven'. Finally, the remaining objects (9 operations, 6 data types, 1 variable, and 1 constant) were deassigned. Figures 1 and 2 show this original Blackjack program and the skeletal structure we created.

Even for this small-scale problem of assigning 17 program objects to 9 packages, a "blind search" technique would generate approximately 8 billion different solution structures. The assistant generated 48 solutions. Upon inspection, we found that 16 of the solutions represented realistic structures. The remaining 32 made semantically improper

[3]Certain Ada constructs such as tasks and generics are not currently supported by the prototype.

package assignments of at least one program object. The runtime performance of the system is satisfactory. The first solution was found after 10 minutes of processing, with the entire problem solving process taking approximately 5 hours on a Texas Instruments Explorer II Lisp Machine.

Figure 3 shows one of the solutions found by the Assistant. This solution can be characterized as making assignments that result in a semantically correct grouping of operations into the "Card" and "Blackjack" packages, without consideration for information hiding issues. As a result, the representation of the 'Ace' constant is visible outside the "Cards" package, because 'Compute_Hand' manipulates it. The representation of the 'Card' type is also visible outside the "Cards" package, because it is manipulated by 'Compute_Hand,' 'Dealer_Plays,' and 'Thorpe_Decide'.

Another solution put the 'Ace' constant into the "Blackjack" package, hiding the manipulation of its representation. The tradeoff of this solution is the inappropriate location of the 'Ace' constant outside of the "Cards" package. Comparing solutions demonstrates an additional benefit of providing a variety of structural alternatives. It makes clear a weakness in the original design of 'Compute_Hand,' and suggests that it should be rewritten to eliminate the manipulation of the internal representation of the 'Ace' constant. A similar redesign of 'Compute_Hand,' 'Dealer_Plays,' and 'Thorpe_Decide' (Blackjack-specific operations) is indicated by their manipulation of the 'Cards' type. Note that none of these issues arise in the initial structure, where a single package holds the program objects to implement both the card deck and the Blackjack game.

6 Discussion

Of course, "objective" data such as the above mean little until far more experience with a range of system structures has been gained. The most useful results we can report at this time have a more subjective flavor and concern our impressions of the power and limitations of this approach.

6.1 Controlling the combinatorial explosion

Controlling the combinatorial explosion in the solution and search space is fundamental to making this approach practical. The two features of the Assistant which address this issue create a problem of their own, however.

First, the assignment heuristics generally allow only a subset of the program object assignments possible for a given partial modularization. While this provides an important means for reducing the search space, it also opens up the possibility of the assistant overlooking good structures because it did not have the heuristics to make the appropriate assignments. A significant issue in the design of assignment heuristics is ensuring that all reasonable assignments can be made without enabling every possible assignment.

The second method for limiting the search space is provided by pruning heuristics.

While over-zealous pruning holds a similar danger to insufficient assignment, it is easier to provide "permissive" pruning criteria which are made more strict as experience with the system grows than to notice that a potentially interesting partial modularization can never be created with the current set of assignment heuristics.

More research is required to understand how to balance the necessity of not expanding the entire search space with the danger of precluding good solutions. A promising direction is in incorporating more knowledge into both assignment and recommendation heuristics—the assignment heuristics could use domain-specific information, while the recommendation heuristics could use information about the progress of problem solving.

6.2 Overcoming cognitive fixation

One control-related problem with the Assistant is a version of "cognitive fixation." The recommendation heuristics prefer more complete partial modularizations, all other things (such as the number of invisible objects, etc.) being equal. The intent of this is to produce a "goal directedness" towards solution states by the Assistant, rather than, for example, performing a breadth-first search across intermediate states. This has an unfortunate side effect, however. If the solution found by the Assistant is sufficiently good, it will then spend a good deal of time "filling in" previously unexplored paths to the same solution, rather than striking off in pursuit of different ones. This corresponds to the classic problem of forcing a hill-climbing algorithm to climb downhill in order to escape a local maximum. It is possible that solutions proposed for hill-climbing algorithms, such as simulated annealing [6], could be incorporated into the Assistant.

6.3 Organization of the solution space

When restructuring small systems, or when using very selective heuristics, the number of solutions returned by the Assistant is small enough that each can be considered individually. In most cases, however, some organization of the the solution space must be provided. The current version of the Assistant organizes the solutions in the order in which they are found, providing a rough "best-first" ordering. Further organization of the solution space could be provided by forming equivalence classes of solution structures. Identification of the characteristics needed to form these classes is a subject for future research.

6.4 Meta-level heuristics

In the current version of the Assistant, the heuristics exist in "one-dimension" in that all of them refer to the application domain of partial modularizations. However, we believe that the performance of the Assistant could be substantially improved by the incorporation of meta-heuristics, or heuristics about the appropriate use and application of domain heuristics. For example, one useful meta-heuristic could provide information

about the utility of the various assignment heuristics. Some of the current assignment heuristics are best thought of as "last resort"—they should be employed only when no other heuristics can be applied to a partial modularization. Other meta-heuristics could dynamically alter the manner in which the recommendation heuristics are combined. Meta-level information like this offers an architectural approach to incorporating facilities to both overcome cognitive fixation as well as prevent unintentionally overlooked partial modularizations.

6.5 Iterative restructuring and reimplementation

Originally, we envisioned the aid of the Assistant as limited to providing a set of alternative system structures for the developer. As we used the system, however, we found that shortcomings in the solutions it generated led fairly directly to fundamental problems in the way the procedures and functions were written. For example, if a function manipulates the internal representation of a type, yet cannot be assigned to the type's package, then there is nothing the Assistant can do to overcome this representation visibility. (This could occur if the type and function had been separated by the developer in the skeletal structure or if the clustering of these two objects led to representation visibility elsewhere in the system.) While enhancing the assistant to rewrite the offending code is beyond its scope, the assistant could attempt to identify such code and bring it to the attention of the developer. In any case, this issue has refined our model of the restructuring process to one in which the Assistant participates in an iterative process of restructuring, identification of problematic code, recoding, and restructuring of the new system.

7 Conclusion

Research on systems like CAPO, KDA, and MIDWIFE indicate an increasing awareness of the importance and utility of automated support for the process of system. However, these systems suffer from the inappropriateness of their metrics for their task, their inability to incorporate domain-dependent structural considerations, and the requirement that the developers accept a single solution from them. This paper describes the Ada Restructuring Assistant, a system which uses the artificial intelligence technique of heuristic search to overcome these problems. The current implementation of the Assistant demonstrates acceptable performance on a small scale problem. More importantly, however, experience with the system has suggested a range of extensions to the original problem-solving strategy, as well as refinements to the restructuring paradigm itself.

8 Acknowledgements

We thank Mike Greenberg for the use of his Grapher program, as well as Dan Corkill and Kevin Gallagher for the use of the Generic Blackboard System.

References

[1] R. Arnold. *Tutorial on Software Restructuring.* IEEE Computer Society, 1986.

[2] G. Booch. Object oriented development. *IEEE Transactions on Software Engineering*, 12(2), February 1986.

[3] E. Bush. The automatic restructuring of COBOL. *Proceedings of the Conference on Software Maintenance*, 1985.

[4] P. Johnson. Inferring software system structure. Technical Report 88-46, University of Massachusetts, Department of Computer and Information Science, April 1988.

[5] J. Karimi and B. Konsynski. An automated software design assistant. *IEEE Transactions on Software Engineering*, SE-14(2), February 1988.

[6] S. Kirkpatrick, D. Gelatt, and M. Vecchi. Optimization by simulated annealing. *Science*, 220, 1983.

[7] H. Sharp. KDA—a tool for automatic design evaluation and refinement using the blackboard model of control. *Proceedings of the Tenth International Conference on Software Engineering*, 1988.

[8] D. Teichroew and E. Hershey. PSL/PSA: A computer-aided technique for structured documentation and analysis of information processing systems. *IEEE Transactions on Software Engineering*, 3(1):41–48, January 1977.

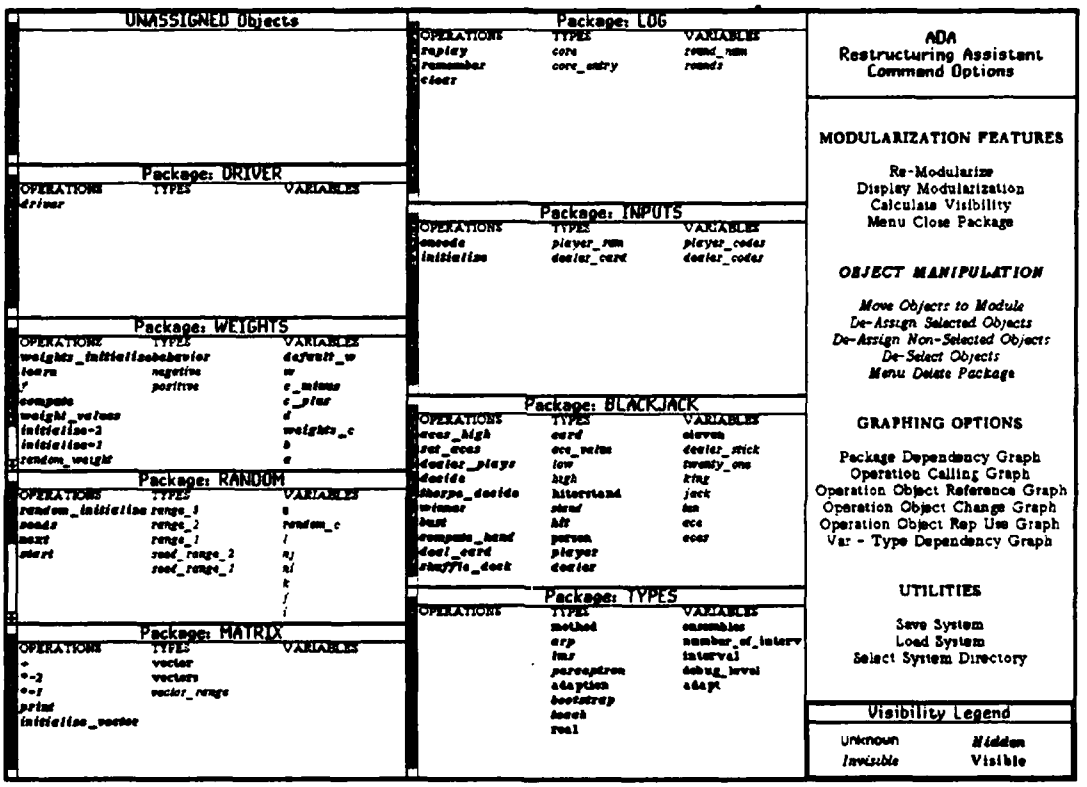


Figure 1: An initial structure.

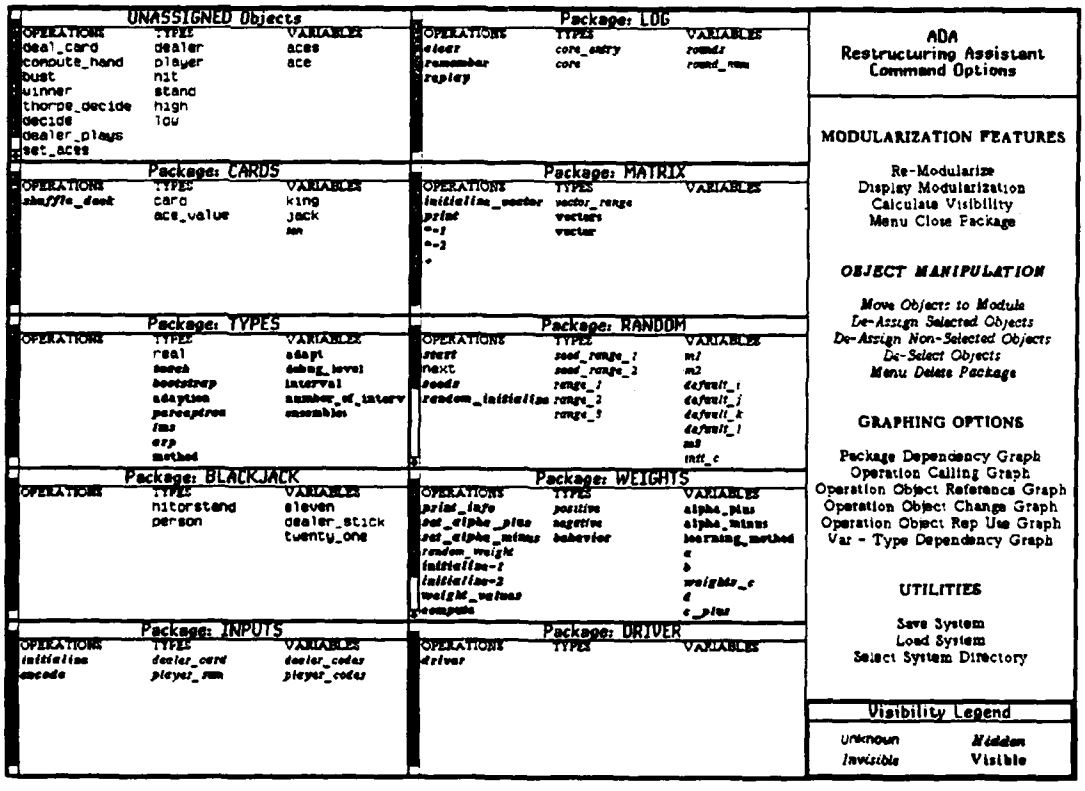


Figure 2: A skeletal structure.

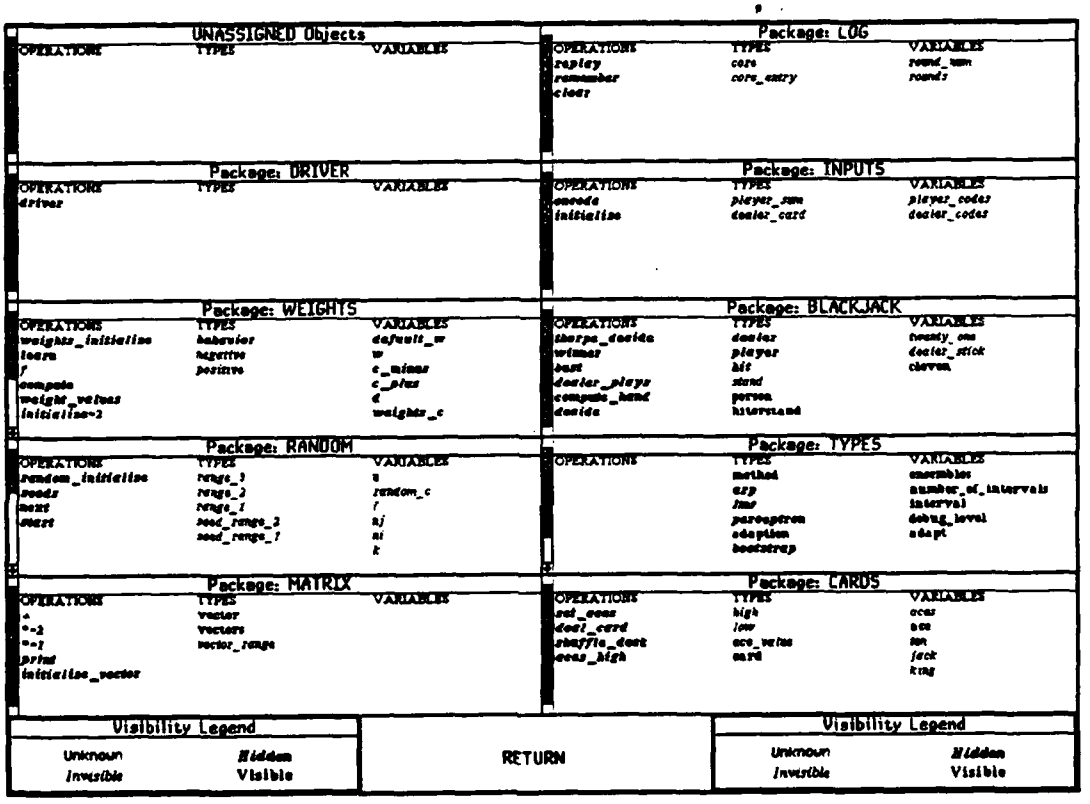


Figure 3: An solution structure.

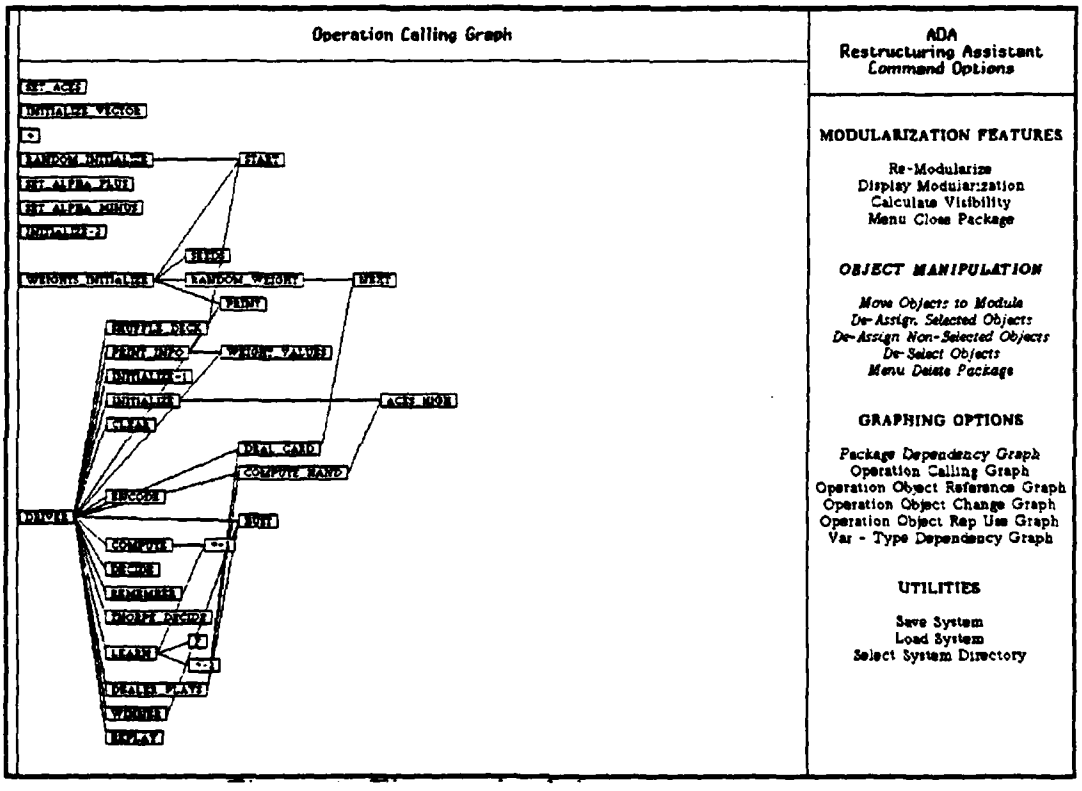


Figure 4: The graph of the calling hierarchy.

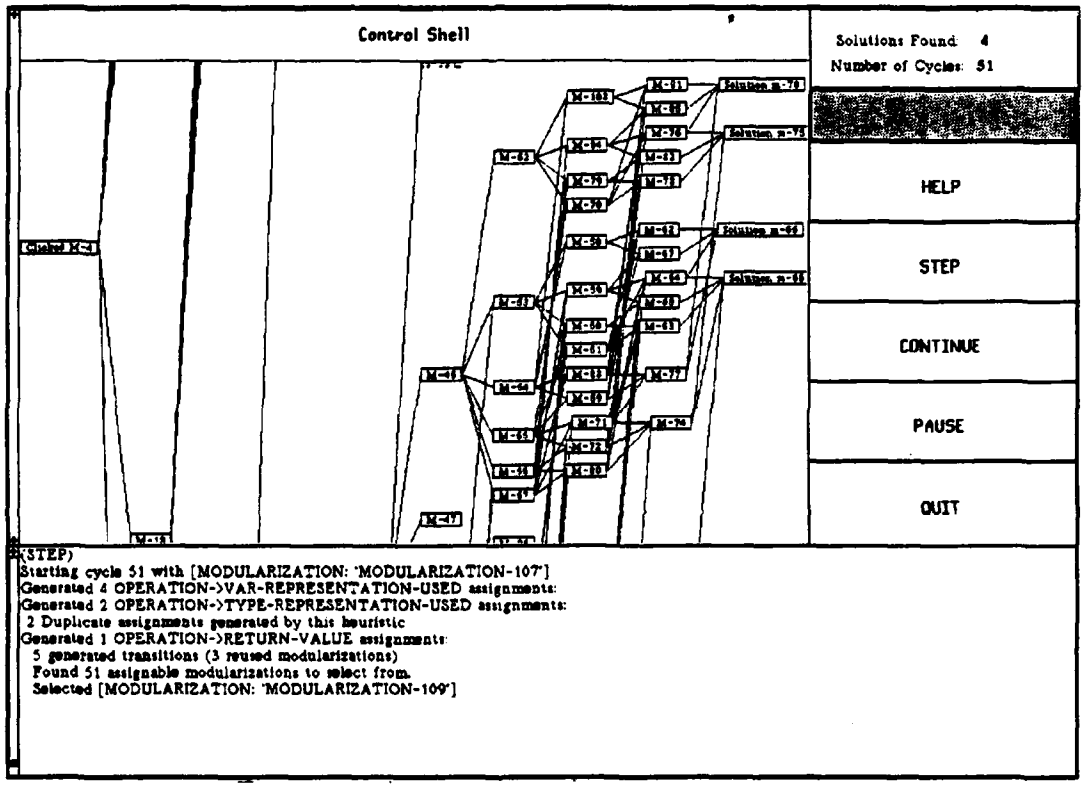


Figure 5: A snapshot of the search process.

Appendix 6-G

An Interface for the Specification of Office Activities

Dirk E. Mahling and W. Bruce Croft

Abstract: An important part of designing an office information system for a particular environment will be the acquisition of descriptions of the activities. We describe a direct manipulation interface, DACRON, for goal oriented specification of office activities. DACRON is based on a recall model of the user's representation of activities. The interface produces plan schemas that are used to construct and execute plans.

This work is supported by the Air Force Systems Command, Rome Air Development Center, Griffiss Air Force Base, New York 13441-5700, the Air Force Office of Scientific Research, Bolling Air Force Base, District of Columbia 20332, under contract F30602-85-C-0008 and Ing.C.Olivetti & C.,S.p.A.

1 Introduction

A number of formalisms have been proposed for the representation of office procedures or, more generally, office work [Zis77,EN80,CL84,WL86]. Most of these formalisms were designed for representational adequacy rather than ease of use. That is, researchers were generally concerned with showing that various aspects of office work could be represented in their formalisms and, subsequently, supported in some way by an office system. The question of whether office workers could use or understand these representations has largely been ignored. The issue, however, is a central one for the development of office systems. Offices are highly dynamic environments and, unless the end users (office workers) can provide descriptions of their activities without the aid of a trained computer person, systems designed to support these activities will simply not be practical. The fact that procedure description languages are often difficult to read is only part of the problem. Graphical presentations such as that used for Officetalk-D [EB82] help in this respect. What is needed is a more integrated approach in which the way users think of their activities (their *recall process*) is incorporated into the design of an interface for specifying and displaying descriptions of their activities.

In this paper, we describe the DACRON interface (Direct Acquisition of Constrained Object Notations) which has three major design objectives;

1. A model of the office worker's conscious knowledge of his activities should be used as the basis for the interface. This model should be tested through experiments with office workers [*MC88*].

2. The interface should emphasize icons, direct manipulation and animation [Shn86,Mah85].

3. The underlying representation formalism for office work is based on *plan schemas* and *constrained objects* [CL87].

The last objective comes from the fact that DACRON will be used as part of the POLYMER environment. POLYMER is a system for supporting office work that constructs plans to achieve office goals. The POLYMER knowledge base contains activity, domain and relation objects organized in generalization hierarchies. Activities are the major components used to construct plans and contain similar knowledge to most plan schemas based on situation calculus. This knowledge consists of a *goal, preconditions, subgoals, effects, and causal and temporal constraints.* A detailed description of the use of this knowledge in the planning process can be found in [CL87]. The domain objects are objects that are manipulated by activities. For example, a purchase form object may be filled out in purchasing activities. Relations are used to represent the semantics of relationships between other objects.

DACRON is used to create an object-based representation of office work and emphasizes the description of the goals of activities. The strong relationship between the recall model used for DACRON and POLYMER'S representation will be shown later in

the paper. In the following section, we present the recall model and some experimental results concerning predictions of the model. We then describe the DACRON interface and how it is based on the recall model. An example of the use of DACRON is given in Section 4. We conclude with a discussion of the work that remains to be done on this project.

2 A Framework for the Recall of Tasks

A designer who intends to build a system that acquires task knowledge from an end user has to carefully identify those aspects of human knowledge he wants the system to acquire and the processes governing its elicitation. He has to distinguish these aspects from those which govern general cognition, interaction with the computer system or language and discourse.

In our case, we would like to acquire knowledge from office workers about the way they execute office tasks. This eliminates existing models of human cognition as possible candidates because those models are too general for our purpose [And83,AB73,Sch82], or they evaluate work done with computer systems themselves [CMN83,PK85], or they address interaction and discourse issues [Suc84,Suc85,Sch75]. Techniques used in knowledge acquisition for expert systems are directly based on existing psychological or linguistic models [Pro86]; no dedicated models for the sole purpose of knowledge acquisition for computer systems were developed.

As specifying a plan can be viewed as writing a highly stereotypical program, we base our work on the general theory of programming [PK83]. This theory identifies four cognitive subprocesses in programming behavior: problem-understanding, problem-solving, coding of solution, debugging.

The *problem* in our case is to give an executable plan for an office task. As we have shown in our experiments, reported later in this section, subjects have no difficulties understanding the *problem*. Solving the *problem* means to an experienced office worker to *recall* appropriate office activities. It is crucial for the interface designer to understand what it actually is that is recalled (the structure of the recalled information) and what processes and factors are involved and influencing the recall of activities, in order to determine if all the information required by the plan formalism can be obtained with this technique. There is a significant difference between the recall of verbal or textbook information and the recall of subject performed tasks [BNC86]. This is another reason (besides the lack of dedicated user models) why knowledge acquisition techniques for rule-based expert systems [Pro86] fail in plan specification for office activities.

The questions of coding and debugging are closely related to the interface and the plan primitives provided and will be discussed in the next section.

As recall is the cardinal method for solving problems of this kind, we will construct a framework for the long-term memory recall of subject performed office tasks, which can be mapped into the plan formalism, to guide the design of a generalized recall method

incorporated into a user interface. We will show which structures and processes can be identified in the recall of activities and which information they convey for the specification of plans.

As we could find no theory that makes statements about structures and processes involved in the recall of complex subject performed tasks from long-term memory, we started a series of interviews and experiments to construct a framework. Asking people how they conduct certain office tasks usually leads to a description of an example. The following transcript of an interview with a secretary illustrates this:

Interviewer: How do you go about buying an item for the office?
Secretary: You mean something small like a paper holder?
Interviewer: Yes, what do you do?
Secretary: Well, first I'd have to find a catalog, an office equipment catalog, that lists the paper holder. When I found it in the catalog, I put down the vendor, the part number, the phone number of the vendor and so on ... all that stuff on the purchase order...
Interviewer: How do you continue?
Secretary: Hmm, I'd call the vendor, they mostly have an 800 number, and ask for the current price...
Interviewer: Hmhm...
Secretary: I'd put the price down on the purchase order, too ... , hmm, and then I'd mail the purchase order to the propriety department ... and I'd have to file a copy of the purchase order in our own books.

In this interview, units of grouped operations can be observed. These units fit very much what is called ***Handlung*** by german psychologists and philosophers. ***Handlung*** is central concept for philosopher like Boesch [Boe80]. In the psychology of sport [Tho78] and the psychology of labor [HR80] it is the basis for human knowledge and activity. A ***Handlung*** is a ***conscious, goal-directed act of a human being, controlled by will, directed towards shaping reality. It contains three aspects: an intended goal, an analysis of means for its achievement and the decision to do so.***[1] A ***Handlung*** contains operations, conducted by a human, which transform states of reality into other states, serving a certain purpose [KB72]. The sequence of ***Handlungen, i.e. Handlungskette***, is what is recalled in the above interview. (Clearly a ***Handlung*** is more than an act but we will refer to it as ***act*** in the remainder of this paper). The act is the smallest coherent unit in the description of a larger activity that appears to be at the appropriate level of abstraction. This individually perceived appropriateness as smallest unit, varying from person to person and from activity to activity, makes the concept very suitable for our purpose. It also

[1]Definition from: Der grosse Brockhaus; 17th Edition. Translation by the authors.

distinguishes the act from GPS-operators [NS72]. While operators are "the rules of the game" as found by a methodological analysis, acts are the representation of the human's perception of these rules. The structure of acts is derived from psychological experiments concerning a task domain. Operators are derived from logical domain analysis. Both the act and the operator share the property of producing new states from old ones in the problem domain, but the evidence for the entity, its appropriateness and its structure differ.

The above definition of acts does not lend itself to immediate operationalization, we try to formalize its aspects in more modern terms [SA77]. The first aspect of an act is, by the above definition, the conscious goal; in our case the intention to complete a certain office task. This intention is equivalent to the statement of the *problem* in the theory of programming. The goal can be regarded as a slot of a larger structure with a specific value. During work on one task, this goal remains the same, only situations change, not intention. People consciously know about this goal and should be ready to report it without difficulty.

The current state of reality is captured in a second slot. We call this slot the pre-situation. It corresponds to the second aspect of the definition of acts. During execution of the same task (same goal) the situation determines which operators are to be applied. Only when the task is completed or interrupted does the goal change. When people are actually performing a task, they directly perceive the pre-situation. In the case of mental simulation, or recall, the content of the working memory mirrors this state. People who are imagining working on a certain task should be readily able to report what the current state of affairs is.

The third aspect of the definition is the decision to generate behavior, which in turn is observable. We represent this as a third slot holding names of the operations to be generated. As we are not concerned with actual execution of tasks, but only the part of them that can be reported by recall, we must be satisfied with the name the person ascribes to a certain operation. The mapping of theses names to primitives of the system is a question of coding and will thus be discussed in the next section.

The result is also available in the recalled act. We call this fourth slot, describing the situation after the application of operations, the post-situation. The post situation is described in the same terms as the pre-situation but it includes the changes caused by the operations. The justification for this slot comes from the explanation that acts transform states of reality into other states of reality.

The complete representation of a formal act is given by the structure in figure 1. We are aware that the operationalization of such a complex concept as an act cannot capture every nuance in meaning and must fall short in certain aspects of description. The structure we present here will certainly have to be amended. Acts are the product of recall of deeper cognitive structures. We assume that recall can be directed to decompose and sequentialize acts. Sequentialization is the process of finding a sequence of acts that lead from one state of the world to another one (*Handlungskette*). Decomposition is the process of breaking an act into lower level acts; making the higher level act the goal

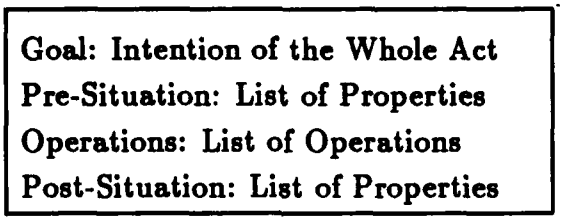


Figure 1: Operationalization for acts

and generating all its constituting operations as lower level acts. As our framework is concerned with the existence of recall processes and structures and not with low level, underlying cognitive processes, we make no assertions on how this is accomplished. It is sufficient for our purposes to know that these processes and structures exist at all, not how and why. This knowledge will lead us hopefully to more sophisticated activity recall techniques than those derived from existing psychological models used in the acquisition of expert knowledge [Pro86]. It is necessary to remember that the specification of subject performed office tasks is not a problem-solving process for the experienced subject because the subject knows the answer already. It is a process of recalling pieces of information that are relevant to planning.

In addition to the recall of activities, we assume the recallability of objects, relations and states of the world. As these entities comply to a certain degree with those items usually encountered in recall experiments, we need to make no additional assumptions.

With the operationalization of acts and the proposed subprocesses, we are able to state the following hypotheses about the recall of subject performed office activities from long-term memory. It is important to realize that these are only one of many possible descriptions for the phenomena occurring but that they seem to be a reasonable one for our purpose.

1. A general task (goal) and a specific situation (pre-situation) will result in the recall of a specific set of operations.

2. Changing the pre-situation or the general task (goal) will result in the recall of different operations.

3. Given a situation, a goal and a later situation will result in the recall of operations and intermediate situations that lead from the earlier situation to the later (*sequentialization*).

4. An operation may itself be composed of acts and those acts are recallable given the operation they form (*decomposition*).

5. Alternative acts are recallable if in a sequence of acts a certain post-situation is said to be not achievable.

6. Operations can be distinguished from the post-situation created by these operations, by identifying properties of their physical or mental environment that have changed.

To test these hypotheses, we conducted a pilot study with four departmental secretaries and a larger series of experiments. Our subjects in those experiments were 74 undergraduate and graduate students at the University of Massachusetts. We could verify all hypothesis but the last one, concerning the post-situation. For a detailed discussion of the experiments and their results see [MC88]. .

3 The DACRON System

DACRON provides users with two major constructs to specify goal-based office activities. The first one is called an *act type* and corresponds to the act of the cognitive model. The act type holds the general description of the goal of the act and the objects, relations, operations and subgoals involved in it. The act type emphasizes our view of the office which is based on goals accomplished through activities, rather than a procedure-oriented description of offices. The second construct is called a *complex type*. Complex types represent objects and relations, which are used in the description of office activities. Complex types correspond to declarative human knowledge. Substituting our act model for the production memory in ACT*, we can say that complex types correspond to nodes in tangled hierarchies. Complex types form an integral part of the act type in the description of office activities.

Both constructs appear in two different forms: the iconic form and the box form (figure 2). The iconic form of the complex type is called *type icon* and the iconic form of

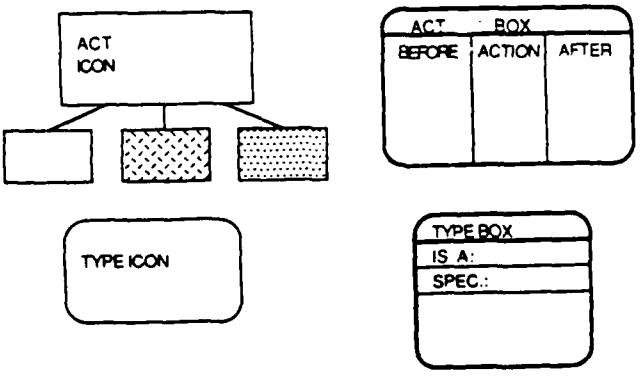


Figure 2: Icon and box representation for act type and complex type

the act type is called *act icon*. The iconic form is the abstract representation of the type. The icon for the complex type differs from that for the act type, as figure 2 shows, in that the act icon has extensions for arguments. Icons carry a label to distinguish between different type icons or act icons.

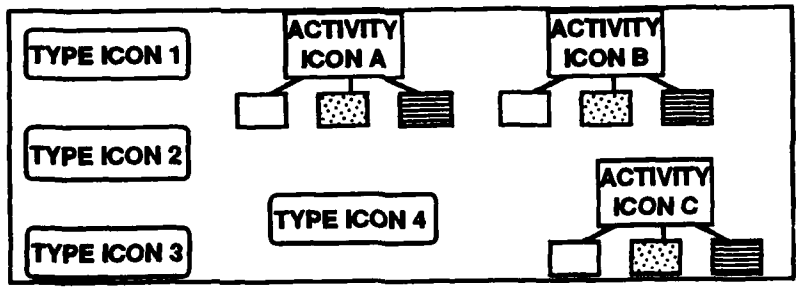


Figure 3: Archive

The box form of the two constructs is also shown in figure 2. We refer to the box representing the complex type as a *type box* and to the box representing the act type as an *act box*. The boxes reveal the detailed specification of the constructs. For this reason, the internal structure of the act box is quite different from that of the type box. The type box shows rows to hold names for arguments and primitive data types for arguments. The act box shows compartments that refer to the pre-situation ("before-compartment"), the operations ("action-compartment") and the post-situation ("after-compartment") of the particular office act. These compartments are ready to accept other icons as input.

The types are by default represented in their iconic form. Only when users wish to see the specification of a construct do they need to employ the box representation.

One of our goals in building DACRON was to establish a relation between the office worker's view of activities and a plan representation. For this reason, both types translate easily into POLYMER's plan formalism. The complex type translates into the object and the relation knowledge structure in POLYMER. The act type translates into the activity knowledge structure. Our model also calls for decomposition of the goal of a general office task into a sequence of acts that appear to the office worker to be at the appropriate level of description. As the granularity of these acts is dependent on the view of the individual, DACRON does not enforce a decomposition hierarchy on the users. The users give the sequence of acts at a level they think is appropriate. The users do this by specifying an act type. Should any component of the act type be unspecified, then DACRON asks the users to specify this component. We coined the term "context-driven" to emphasize the synthesis of the goal-based character of the task and the data-driven determination (pre-situation) of the next action. The human view of an office task is not goal-driven or data-driven alone, but looks at it as serving a main goal, with the sequence of acts taken depending on the goal *and* the current situation.

Every item in the knowledge base is represented by an icon. The users have a viewport into the knowledge base called the *archive*. This archive is a window that is always on the screen. Users are allowed to move and rescale the archive window, but not to close it. At any given moment the archive shows a fixed part of the knowledge base (figure 3). To see different parts of the knowledge base, the archive-viewport can be moved over the knowledge base, which is represented as a two-dimensional plane filled with icons. Users may in addition zoom in and out on the knowledge base.

To reorganize the clustering of icons in the knowledge base, users may employ the *finder*. The finder allows users to order the icons in the knowledge base by certain keys like name, date of creation, etc. or to search for icons by goal, constituent parts, etc.. These finder commands can be invoked by opening a special icon that is always present in the archive, the *information stand*. Upon opening this icon, users can pick commands from a menu table and provide the desired options. An icon may be copied from the archive to the *layouter* or the *clipboard*. The clipboard is merely a buffer that holds frequently-used icons. It serves as the system's counterpart of the users' working memory. Icons may be moved from the clipboard to the archive or the layouter and vice versa. The layouter is DACRON's workbench. It can be viewed as a constantly visible text and graphic screen editor. Only in the layouter can icons be opened to inspect their box representation. The modification and specification of act types and complex types takes place in the layouter. The specification process depends on the type. Act types are specified by moving appropriate icons into the "before compartment" (pre-situation), the "action compartment" (operations), and the "after compartment"(post-situation) of the act box. Complex types are specified by typing names for arguments and attaching appropriate icons representing primitive types to these names.

DACRON has two other facilities. A *debugger* can be invoked to give an animated demonstration of the specification process of a type. The animation will show how the type was created, in which order arguments were given, what icons were placed into the type, etc. As this debugger is interactive, it is possible to stop it at any given moment and change the specification of the type shown. The animation can also be used for training purposes, teaching new users how DACRON is used. The second facility is the *reviewer*. It is used to present plan networks and alternatives at crucial points in the planning process to the users. In this situation, we use the DACRON interface not only for the specification of activities, but also for the presentation of the planning process in POLYMER. The architecture of DACRON is shown in figure 4.

All manipulations on icons are executed with a pointing device (mouse) and the keyboard. The mouse is used to select icons, copy them, move and place them. To select the size and location of one of the three windows on the screen (archive, layouter and clipboard) we also employ the mouse.

In order to create entirely new act types or new complex types, an empty icon for the type must be copied into the layouter, opened to show the box representation, which is empty in this case, and typing the name for the new type in the header of the box. Then the content of the type can be specified. Users type input for argument names in the rows of the box. They then move a primitive type, specifying the data type behind it. Primitive types, like integer, real, and string are available as icons in the archive. Two named rows (IS-A and SPECIALIZATION) appear under the header of the type box. Constraints may be put on the complex types by opening the primitive type icon behind the argument name and selecting values. We also allow for the specification of set and relational constraints. We will illustrate the techniques employed for this in the example

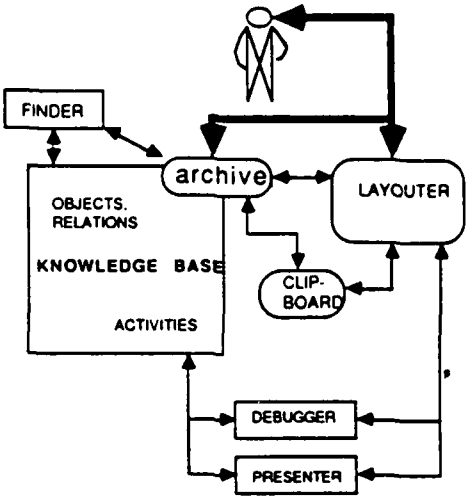


Figure 4: DACRON architecture

in the next section. To specify an act, users place and constrain complex type icons and other act icons in the compartments of the act box.

4 An Example

In this example we give a partial description of how the process of purchasing an item would be specified in DACRON. Figure 5 shows the complete plan diagram for the purchasing activity as extracted from an interview with a departmental secretary. The leaves in this tree are assumed to be plan primitives available in the plan library. All other nodes have to be constructed using DACRON.

We start by copying an empty act icon with the mouse from the archive to the layouter. Using the keyboard, we give it a name. As the icon is still undefined, it appears with dashed rather than solid borders. Figure 6 shows the copied act icon and the archive. In the archive we can see plan primitives represented as act icons (MAIL, INPUT, COPY), some object icons (DEPARTMENT, CHECKBOOK), primitive type icons (INTEGER) and the empty object icon and act icon we copied. In order to specify the act type, we have to open the "purchase-an-item" icon with the mouse. Figure 7 shows the box corresponding to the icon. It is empty because we have not specified it yet. It will stay dashed until we complete the specification process to remind us of this fact.

Now we have to tell DACRON what we do to purchase an item: Fill-out-purchase-order, File-a-copy, Mail-purchase-order, Receive-shipment, Pay-invoice. These are our acts in doing the task of purchasing; it is our individual perception of how to accomplish the task. We start "telling" DACRON by moving act icons in the actions compartment of the "purchase-an-item" box. In the case of existing act icons or plan primitives (repre-

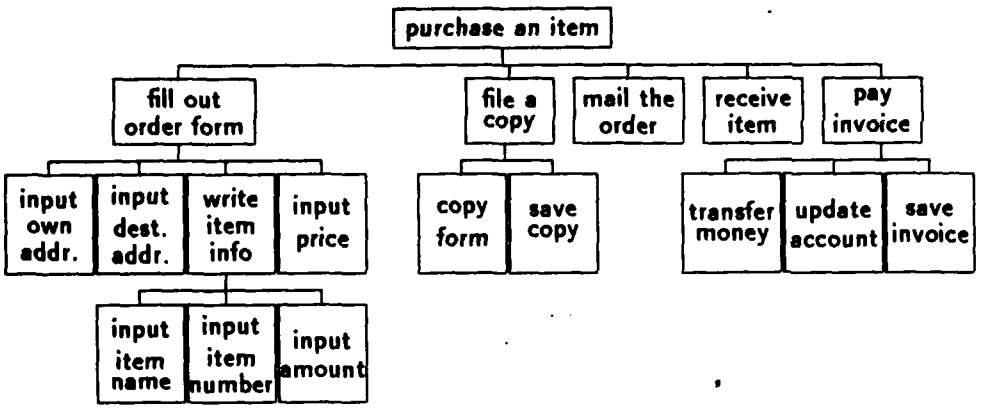


Figure 5: Plan for Purchasing an Item.

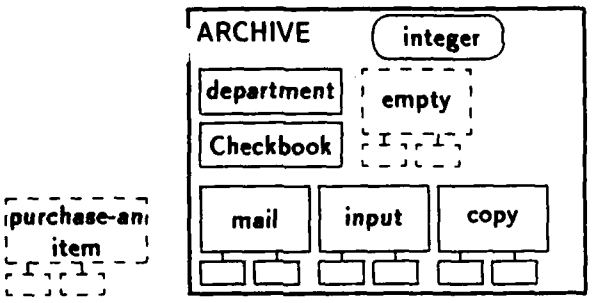


Figure 6: Copying empty Act Icon from the Archive.

sented as act icons) this is fairly simple. We just look them up in the archive, copy them and move the copy to the actions compartment. If we cannot find an existing act icon we need, we have to create it just as we created the "purchase-an-item" icon. We copy the empty act icon, name it and move it to the actions compartment. Note that these act types are not specified but the existing ones we found are. This distinction can be seen in figure 8 where "Mail" and "Receive" are solid, indicating specified types and the others are dashed, indicating unspecified types.

Before we start specifying these new act types, we decide to do more work on the current act: We want to specify the preconditions, the state of the world that allows us to apply this act. This situation is tied to objects and relations in the world. What is this situation in our case?

- we have to have a purchase order.
- we have to have a checkbook.
- there must be more money in the account than the item costs.

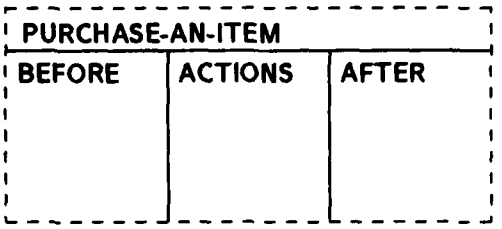


Figure 7: Dashed Open Act Box.

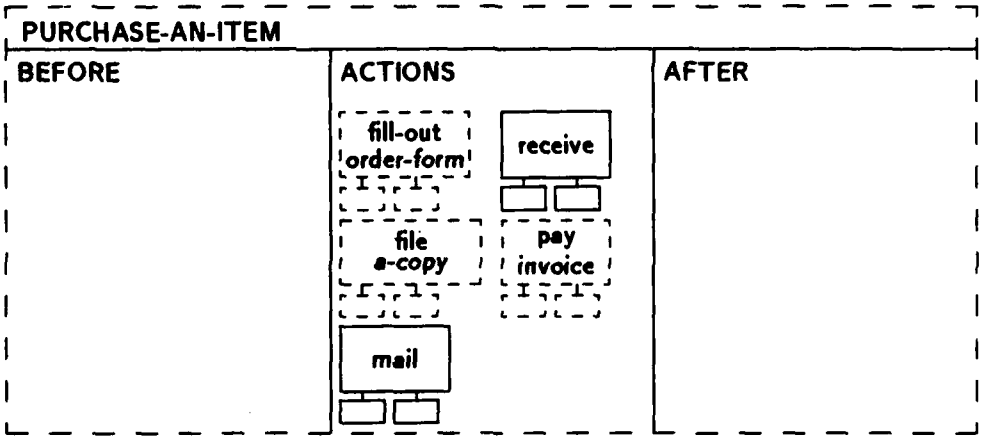


Figure 8: Act Icons in "Actions" Compartment.

Employing the finder, we look for a purchase-order icon in the archive. In this case the finder only returns a general icon for forms. We open it and recognize that it has some of the attributes we need in a purchase order. We decide to use this "Form" object to build our purchase order as a specialization of it. We copy an empty object icon, name it "purchase-order," open it and copy the "Form" icon to the "Is-a" row, indicating that the purchase order is a form. The two left boxes in figure 9 summarize this situation and show the definition of the form in the box next to the box to be defined purchase-order on the very left.

All attributes of the "Form" are inherited and displayed (not shown in the figure). Just like the act box and the icons, this type box is dashed to indicated that its specification is not yet completed. Using the keyboard we add more attribute rows to the box to make the purchase order a true specialization of the form. To specify the data types of the attributes we copy primitive type icons from the archive to the slots behind the attribute names. Figure 9 shows the "PURCHASE-ORDER" at the very right box with the inherited attributes from the "FORM" and the attributes added.

As the finder readily returns the checkbook, we have all the objects to put in the

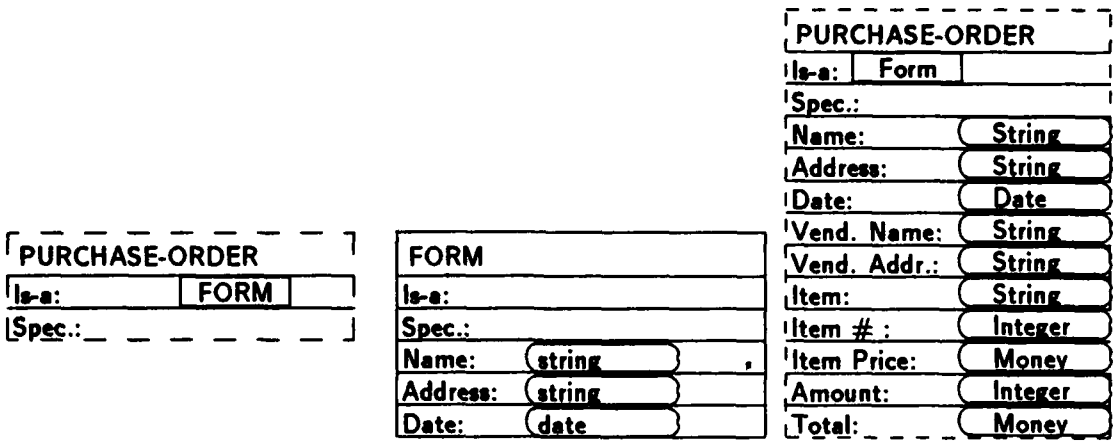


Figure 9: Purchase Order in Initial Stage, Form and Completed Purchase Order.

"before compartment." We copy a "purchase-order" and a "checkbook" icon right into the compartment. Now we need to tell the system that there has to be more money in the checkbook than the item will cost. As this concerns only the two icons in the before compartment, we can operate directly on them. We open both of them and copy the "price" row of the purchase order and the "balance" row of the checkbook. We stick the two rows together and DACRON automatically adds scroll menus with operators and values to specify relational and interelational constraints (figure 10). Scrolling the menu between the rows and selecting the "<" operator, we specify the "item costs less than balance in checkbook" precondition. By closing all the open boxes we end the precondition specification and arrive at the situation depicted in figure 11.

Let us summarize what we have achieved so far:

- we have a list of acts in the actions compartment.
- some of these acts need further specification.
- we have all preconditions specified in the before compartment.
- all objects we need at this level are in the before compartment.

Now we want to work on the unspecified acts in the action compartment. They are easily identified as they are dashed. We open the "Fill-out-order-form" icon. In the cognitive model this would be viewed as decomposition, invoking another sequence of acts. DACRON gives us another dashed, open act box. Using the same procedures as we did in specifying the "purchase-an-item" act, we start specifying the "Fill-out-order-form" act. Figure 12 shows us a stage in the specification of this act. Note that we use only one unspecified act, which at the next level down can totally be specified by primitives,

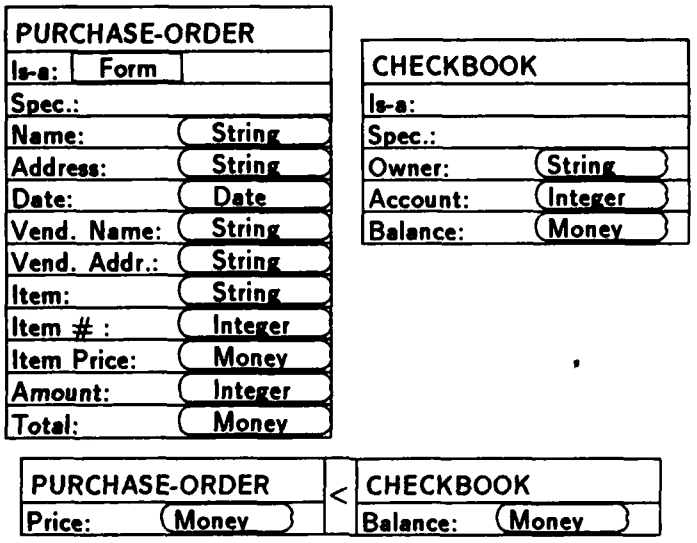


Figure 10: Specifying a Constraint

following the diagram in figure 5. Eventually all dashed act icons in the "purchase-an-item" box are specified and solid. All lower level acts are completely specified in the same manner as the highest level act ("PURCHASE-AN-ITEM") we use as an example.

Now it is time to specify the effects of this act in the "after" compartment. The effect of our act is that we, the department, possess the item. Note that the effect of having less money would have been specified in the "pay" act. Employing the finder we look up the "possess" relation in the archive (if it did not exist already we could easily create it), copy it to the "after" compartment and open it. We open the primitive icon for strings in the "possessor" row and type "University of Mass." We also have to make sure that the item we talk about here is the same we purchased. So we copy the "item-name" row from the purchase order in the "before" compartment to the "after" compartment and join it with a copy of the "item-name" from the "possess" relation. The constraint definition works in the same way as the constraint definition for the "before" compartment (figure 10). The completely specified "purchase-an-item" act can be seen in figure 13.

5 Future Work

We currently have a shell of the DACRON system implemented. The next implementation step will involve the specification of constraints. Later work will be devoted to the animated presentation of planning processes and the use of animation in debugging the description of activities. We also intend to develop a more appropriate representation for the relation knowledge representation structure in POLYMER. It appears that the use of color coding techniques has promise for constraint definition and especially for the

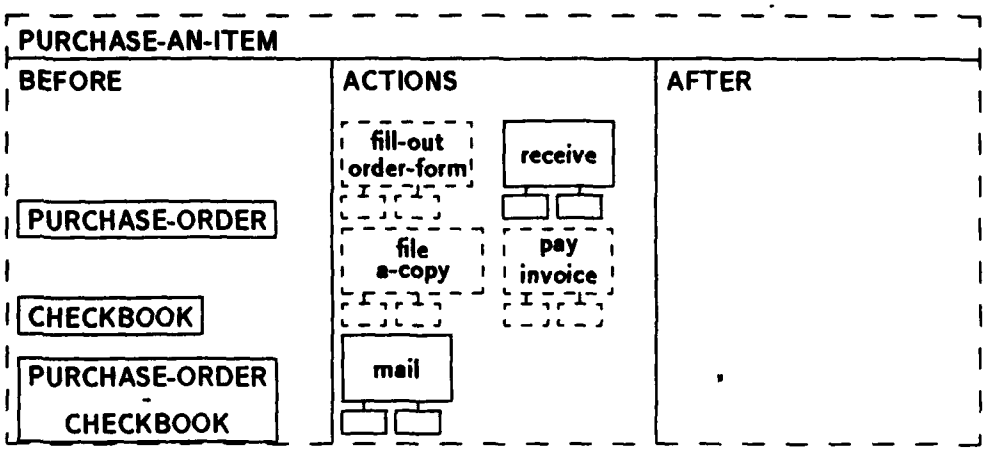


Figure 11: Purchase Act with Actions and "Before" Situation.

specification of logical expressions as used in domain constraints by marking regions of valid values.

We are planning to design user studies with a prototype of the DACRON system in an office environment.

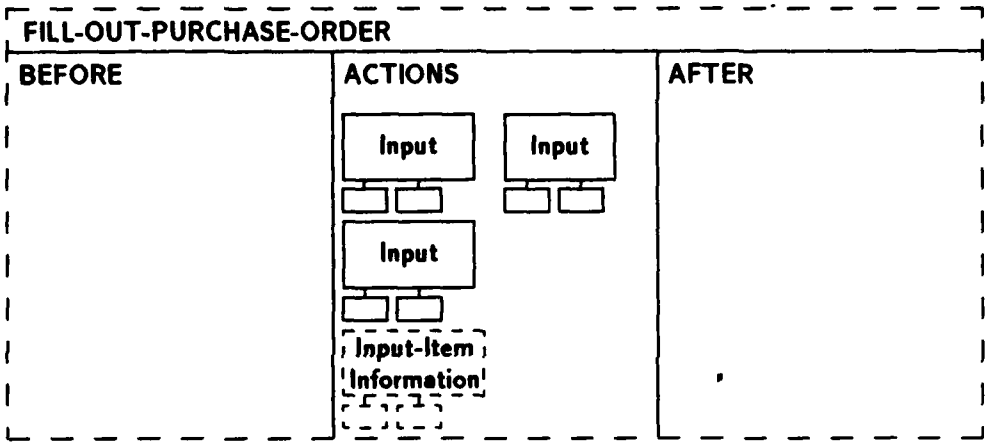


Figure 12: Specifying the Subgoal Decomposition.

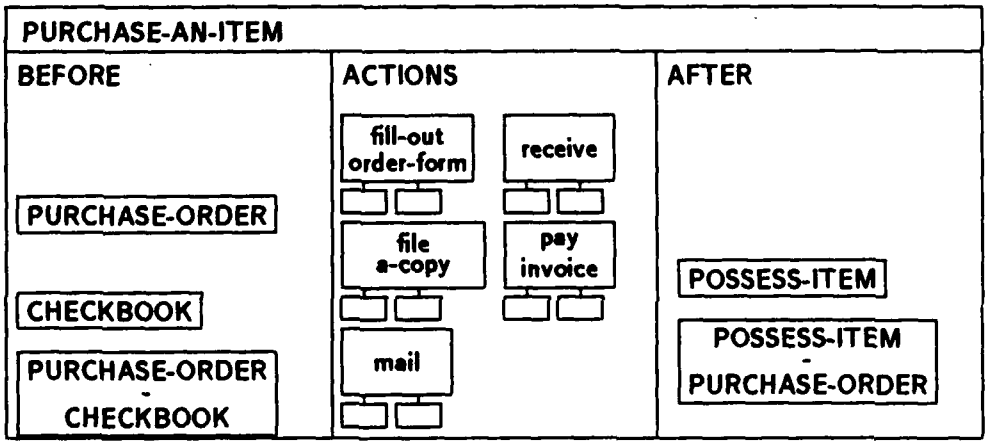


Figure 13: Completely Specified Act.

References

[AB73] John R. Anderson and Gordon H. Bower. *Human Associative Memory.* Winston and Sons, 1973.

[And83] John Robert Anderson. *Architecture of Cognition.* Harvard University Press, 1983.

[BNC86] Lars Baeckmann, Lars-Goeran Nillson, and David Chalom. New evidence on the encoding of action events. *Memory and Cognition*, 14(4):339–346, 1986.

[Boe80] E. E. Boesch. *Kultur und Handlung.* Bern, 1980.

[CL84] Bruce Croft and Larry Lefkowitz. Task support in an office system. *ACM Transaction on Office Systems*, 2:197–212, 1984.

[CL87] Bruce Croft and Larry Lefkowitz. A goal-based representation of office work. In *Proceedings of the IFIP Workshop on Office Knowledge*, North Holland, Toronto, 1987.

[CMN83] Stuart K. Card, Thomas P. Moran, and Allen Newell. *The Psychology of Human Computer Interaction.* Lawrence Erlbaum, 1983.

[EB82] C. Ellis and M. Bernal. Officetalk-D: An Experimental Office Information System. In *Proceedings of the first ACM SIGOA*, pages 131–140, 1982.

[EN80] C. Ellis and G. Nutt. Office information systems and computer science. *ACM Computing Surveys*, 1:27–59, 1980.

[HR80] W. Hacker and H. Raum. *Optimierung von kognitiven Arbeitsanforderungen.* VEB Deutscher Verlag der Wissenschaften, Ost-Berlin, GDR, 1980.

[KB72] G. Klaus and M. Buhr. *Philosophishes Woerterbuch.* VEB Deutscher Verlag der Wissenschaften, Ost-Berlin, GDR, 1972.

[Mah85] Dirk E. Mahling. *Beschreibung und Bewertung von Fenstertechniken in der Dialoggestaltung nach kognitiv-ergonomischen Kriterien.* Master's thesis, Technische Universitaet Braunschweig, Braunschweig, West Germany, 1985.

[MC88] Dirk E. Mahling and W. Bruce Croft. *A Conscious Human Task Representation Model for Plan Specification.* Technical Report, University of Massachusetts at Amherst, 1988. to be released.

[NS72] Allen Newell and Herbert Simon. *Human Problem Solving.* Prentice-Hall, 1972.

[PK83] R. D. Pea and D. M. Kurland. *On the Cognitive Prerequisites of Computer Programming.* Technical Report TR 18, Bank Street College New York, 1983.

[PK85] P. G. Polson and D. E. Kieras. A quantitative model of the learning and performance of text editing knowledge. In L. Borman and B. Curtis, editors, *Human Factors in Computing; CHI '85*, ACM, Inc., New York, 1985.

[Pro86] American Association for Artificial Intelligence. *Proceedings of the Knowledge Acquisition for Knowledge-Based Systems Workshop*, Banff, Alberta, Canada, November 1986.

[SA77] R. C. Schank and R. P. Abelson. *Scripts, Plans, Goals and Understanding.* Lawrence Erlbaum, 1977.

[Sch75] C. F. Schmidt. Understanding human actions. In *Conference on Theoretical Issues in Natural Language Processing*, 1975.

[Sch82] Roger C. Schank. *Dynamic Memory.* Cambridge University Press, 1982.

[Shn86] Ben Shneiderman. *Designing the User Interface: Strategies for Human Computer Interaction.* Addison Wesley, 1986.

[Suc84] Lucy A. Suchman. Office procedures as practical action. *ACM Transaction on Office Information Systems*, 4:320–328, 1984.

[Suc85] Lucy A. Suchman. *Plans and Situated Actions: The Problem of Human-Machine Communiation.* Technical Report isl-6, Xerox Corporation, 1985.

[Tho78] A. Thomas. *Einfuehrung in die Sportpsychologie.* Goettingen, Toronto, Zuerich, 1978.

[WL86] C. C. Woo and F. H. Lochowsky. Supporting distributed office problem solving in organizations. *ACM Transactions on Office Information Systems*, 4:185–204, 1986.

[Zis77] M. Zisman. *Representation, Specification and Automation of Office Procedures.* PhD thesis, Wharton School, University of Pennsylvania, 1977.

Appendix 6-H

Working with Expert Teachers to Distinguish Knowledge from Theory

Beverly Woolf
Klaus Schultz†
Tom Murray

†School of Education
University of Massachusetts
Amherst, Mass 01003

Abstract: We have studied how domain and teaching experts help students understand difficult concepts in science. We model both the scientist's view of selected physics problems and a teacher's knowledge about how to learn and tutor these same problems. This paper presents a framework for representing and acquiring such knowledge in preparation for building an intelligent tutor. An AI modeling methodology is used to represent cognitive processing, teaching expertise, and scientific reasoning. The framework is domain-independent and has been used to design and implement several tutors. Two such tutors are described.

This work was supported in part by the National Science Foundation, MDR-8751362, Air Force Systems Command, Rome Air Development Center, Griffiss AFB, New York, 13441 and the Air Force Office of Scientific Research, Bolling AFB, DC 20332 under contract No. F30602-85-C-0008. This contract supports the Northeast Artificial Intelligence Consortium (NAIC). Partial support also from an ONR University Research Initiative Contract No. N00014-86-K-0764.

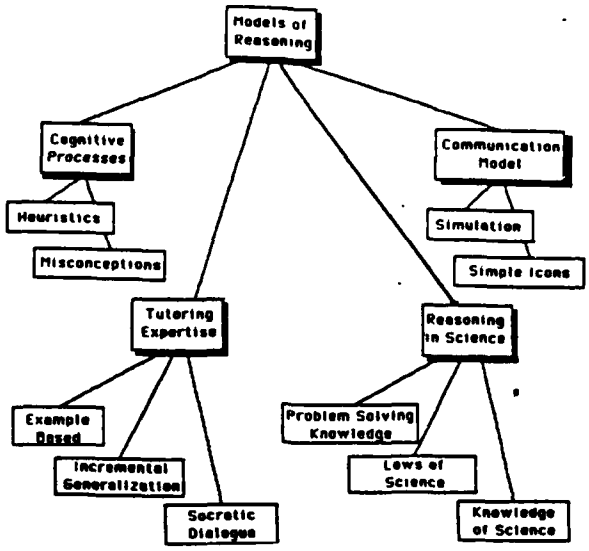


Figure 1: Models of Reasoning in an Intelligent Tutoring System (after Clancey[1987])

1 Working with Expert Teachers

Research results from cognitive scientists, domain experts, and educators provide information about how a student learns, the differences between novice and expert problem solvers, and key parameters of the student model. Such results are necessary before truly effective and responsive intelligent tutors can be built. Building student models and teaching strategies into such systems requires both explicit and implicit encoding of large amounts of information. It requires information about how the system will make assumptions about the student, hold beliefs about his/her knowledge, and impart propaedeutic principles (the knowledge needed before learning about a domain, such as learning vector analysis in order to study mechanics) [Halff, to appear].

Figure 1 shows some of the models of reasoning that we encode in each of the intelligent tutors we build. Each model requires probing the relevant experts with specially designed questions. In this paper, we focus on how to elicit knowledge for three of these models: cognitive processes of the student, science expertise, and tutoring expertise. We describe techniques used to acquire and update such knowledge and indicate how successful we have been in working with experts to encode their knowledge into tutoring systems.

As an illustration of the kind of primitive knowledge we try to elicit from the research literature and from our experts, we list some components of the cognitive processes model that we hope to acquire:

Student's Underlying Knowledge:

Is the domain seen by the student as fragmentary, coherent, or interpretable?

Is the student confused, bored, or knowledgeable?

Do novices ask for definitional information or can they process the task using their own knowledge?

Heuristics used by a Student:

Rules of thumb used to solve problems in the domain (e.g., examine an extreme case; look at a simple case).

Ideal Behavior:

Steps taken for each task.

Self-diagnosis:

Does student recognize his/her cognitive difficulties?

Is student insecure or uncertain about how to apply knowledge?

Common Errors and Misconceptions:

Identify ways to correct misconceptions.

Identify errors and ways to diagnose them.

Identify misconceptions and ways to tease misconceptions apart.

Inferring Goals. An important feature of intelligent tutoring is to infer goals and *intentions* about a user from observed behavior. For tutoring especially, it is important that a user's evolving "mental state" during the course of the dialogue be made explicit. Thus a representation of the user contains not only explicit stored knowledge, but also should implicitly update that knowledge as the dialogue continues [Kass & Finin, 1987]. This kind of dynamic modeling might be less necessary in other systems, such as text generation.

Models of Reasoning. We use an AI modeling methodology to identify primitives for reasoning about tutoring and student knowledge. The control and knowledge structures we have developed are **explicit, reusable**, and **generative** [Clancey, 1987]. They are general at the domain level, i.e., structures have been used to build several tutors and are transferable to new domains. The models are also generative, showing generality at the case level, i.e., procedures can be used again and again to solve new problems in a single domain. The resulting mechanisms also **promote experimentation** in that continued testing of systems allows us to learn more about the applicability of an individual model and how to fine-tune its reasoning mechanism.

The models we have developed fit into a framework which includes the questions used to extract primitive information from experts. For each model, we identify questions focused so that their answers fit specific areas of our system's knowledge or control mechanism. In this paper we enumerate those questions and provide some clarification of the

expected answers. Where appropriate, we indicate how and where we stored the answers.

Physics Theory is not Knowledge. We have been working with physics teachers to encode their knowledge about teaching and learning physics. Physics theory includes many principles, rules, and formulas used to analyze problems in physics. However, these principles, rules, and formulas do not constitute all the relevant knowledge of a domain. Teaching knowledge includes, for instance, knowledge about when certain principles and rules are appropriate to invoke, how they might be applied, etc. [Clancey, 1987].

Some teachers, when questioned about how they solve statics problems, enumerated formulas. But formulas are not enough. We need knowledge in the more general sense indicated above: information about how students identify variables in the problem; how they choose equations, reason about situations, identify attributes of a situation; and methods they use to classify problems into known solution types.

In fact, modeling problem-solving processes requires focusing on how and why a scientist asks questions, proposes experiments, or uses heuristics (see Section 3) while solving problems. Acquiring such knowledge from experts is not easy. For example, in building the statics and thermodynamics tutors (described in Section 2) we worked with cognitive scientists, physicists, astronomers, and potential users of the system, such as high school and college teachers and students, for more than 18 months. We produced over 100 pages describing processes, screen designs (including help activities about physics), and cognitive studies (identifying educational goals, potential errors, and misconceptions) before any code was built.

2 Case Examples: Building Tutors For Statics and Thermodynamics

We have built two tutors in conjunction with the Exploring Systems Earth (ESE) Consortium [Duckworth et al., 1987].[1] These tutors are based on interactive simulations that encourage students to work with "elements" of physics, such as mass, acceleration, and force. The goal is to help students generate hypotheses as necessary precursors to expanding their own intuitions. Students develop models of the physical world, while discovering and "listening to" their own scientific intuitions. Identifying the cognitive processing primitives listed in Sections 1 and 3 begins to allow us to track the student's cognitive processes as he/she develops these hypotheses.

In these tutors, all interactions with the tutor, including questions asked, responses given, and requests made, are used to formulate the system's next teaching activity and strategy. These tutors have been described elsewhere (see for example, Woolf et al., 1988; Woolf & Cunningham, 1987; Woolf & Murray, 1987) and will only be summarized here.

[1]ESE is a group of universities and industries working together to build intelligent science tutors. The schools include the University of Massachusetts, San Francisco State University, and the University of

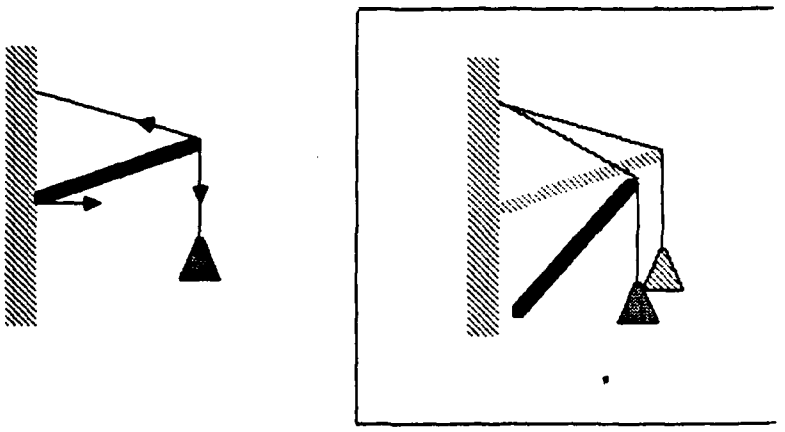


Figure 2: Statics Tutor: Interface and Simulation

Figure 2 shows a simulation for teaching concepts in introductory statics. In this example, students are asked to identify forces and torques on the crane boom, or horizontal bar. When the beam is in static equilibrium there will be no net force or torque on any part of it. Students are asked to draw appropriate force vectors to solve the problem.

If a student were to specify the forces indicated by the heavy vectors in Figure 2 A, the solution would be incomplete: a force vector is missing (located at the wall and pointing upwards).

There are many possible responses to such a student error: the tutor might present an explanation, provide another problem, or demonstrate that the equations resulting from the specified forces cannot be solved. Still another response would be to demonstrate the consequence of omitting the "missing" force: i.e., the end of the beam next to the wall would crash down. The system selects this latter action because we want to show students how their conceptions and the observable world might be in conflict and to help them visualize both their internal conceptualization and the science theory. This last response is selected by a discourse management mechanism that facilitates context-dependent machine responses [Woolf & Murray, 1987].

A second tutor is designed to improve a student's intuition about concepts such as energy, energy density, entropy, and equilibrium in thermodynamics by making use of a very simplified but instructive simulated world consisting of a two-dimensional array of identical atoms (Figure 3 [Atkins, 1982]).

Like the statics tutor, the thermodynamics tutor monitors and advises students about their activities and provides examples, analogies, or explanations based on student actions, questions, or responses. In this simplified world the atoms have only one excited

Hawaii and industries include the Hewlett-Packard Corporation.

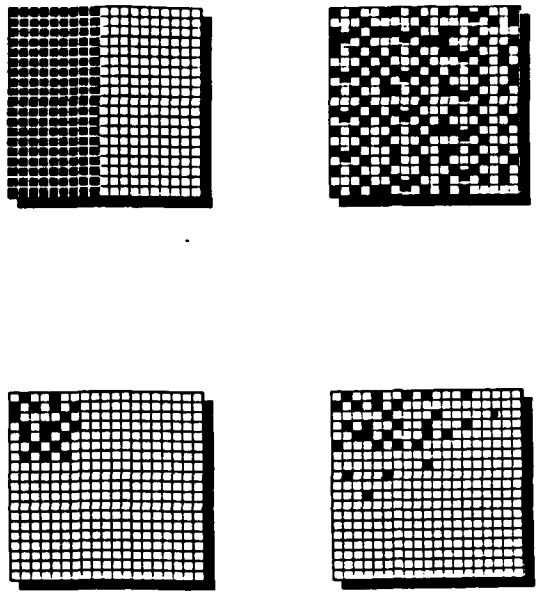


Figure 3: Thermodynamics Tutor: Simulation

state; the excitation energy is transferred to neighboring atoms through random collisions. Students can specify initial conditions, such as which atoms will be excited and which will remain in the ground state. They can observe the exchange of excitation energy between atoms and can monitor the flow of internal energy from one part of the system to another as the system moves toward equilibrium. In this way, several systems can be constructed, each with specific areas of excitation. For each system, regions can be defined and physical qualities, such as energy density or entropy, can be plotted as functions of time.

3 Modeling Students' Cognitive Processes

We prefer that prior cognitive research in the domain be available to the designers of intelligent tutors. Existing studies ought to provide information about how students learn in the domain, the differences between novice and expert problem solvers, and key parameters of the student model (e.g., Anderson [1981] and Woolf et al. [1986]). For example, before the statics tutor was built, more than a decade of prior research into physics misconceptions in mechanics was conducted by other researchers (e.g., McDermott [1984] and Clement [1982]). This knowledge was reviewed and included in the statics tutor.

Identifying Heuristics. Cognitive knowledge is not easily identified, nor quickly provided by experts. For instance, we asked physics experts to provide heuristics for solving

physics problems. Heuristics are not the actual steps used for solving problems.[2] Rather, heuristics are the "rules of thumb" that any person, expert or novice, uses to approach a problem. In the simulation environment, heuristics include "first try a simpler version" and "relate to your everyday experience." Such rules are not guaranteed to work; they are only intended to reduce the search.

To facilitate teaching these heuristics, the machine tutors we build allow the learner to:

1. *inspect* systems in action;
2. *manipulate* parameters, such as size of universe, density of excited states vs. those at ground state;
3. *answer qualitative questions* using
   (a) *prediction*
   (b) *comparison*
   (c) *modification*;
4. *focus* on one topic at a time, figure out which parameters to change and what to look for;
5. *incrementally* change the system to compare cases that are similar in all but one respect.

We encourage use of such heuristics by allowing students to manipulate elements of the simulation. For instance, in the thermodynamics simulation students can manipulate the size of the universe, the number of atoms at excited or grounded state, and the number of observation windows. In the statics tutor, they can manipulate elements such as the size of the boom, the mass of the weight, and the angle of the boom.

The tutor can lead the learner through tactical steps to experiment with these parameters or it can engage him/her in a kind of "Socratic" dialogue. Regardless of whether the initiative rests with the tutor or the learner or whether there is mixed-initiative dialogue, it is important that such heuristic capabilities be explicitly built into the tutoring system.

Student Misconceptions. Research into cognition, particularly into the learning of physical concepts, has yielded much information about misconceptions [McDermott, 1984; Clement, 1982]. Misconceptions is the technical term used to describe student conceptions at variance with standard current understanding of science and which are rooted in

[2]For example, to solve a typical problem in statics, one identifies forces acting in the situation, represents these in vector diagrams, writes algebraic expressions based on these diagrams and the physics principles (e.g., vector sums of the forces and the torques are each zero), and then solves the equations for the unknown quantities.

students' intuitions and everyday experiences. Many such misconceptions are very persistent in the face of even very good and very focused teaching of the correct conceptions and are very common even in students who achieve well in standard science courses. Fortunately, these common and "deep" misconceptions are finite in number. Research has begun to identify ways to help students overcome these misconceptions [Clement, 1982]. The domain in which the most work has been done is classical mechanics, of which statics is a part. We have used research results on diagnosing and remediating misconceptions in our tutor design.

When a student makes an error, it need not be due to a misconception: errors may be due to oversight or failure of memory, lack of necessary information, failure in the mechanics of equation solving, uncertainty in how to apply "known" principles in the particular example, as well as to genuine misconceptions. An intelligent tutor must make suppositions as to the cause of an error, test these, and respond accordingly. Some examples from the crane boom problem follow.

- A student may fail to include in a listing or diagram of forces acting on the boom the force exerted by the wall on the boom. This may be an indication of a common misconception: the belief that static, rigid objects such as walls cannot exert contact forces–they can only "be in the way" and prevent the other object from moving.

- A student may include a force at the point where the boom touches the wall, but draw the force vector *into* the wall rather than away from the wall. This may indicate a *confusion in the student's mind* between forces exerted *on* the body in question and forces exerted *by* it.

- A student may correctly include the force of gravity on the boom, but place the effective point where this force acts not at the center of mass but at the end of the boom. This may indicate that the student lacks knowledge about the effective point of action of gravity on an extended body, or that the perceptual salience of the end of the boom is strong enough to overcome what he/she has "learned" in a general way about gravitational forces.

Other misconceptions have been identified in the area of thermodynamics, a difficult domain to understand. The underlying knowledge is hard to visualize and causes problems even for engineering and science majors and those who breeze through the study of statics.

One misconception might be to confuse heat and temperature. Another concerns equilibrium. Most people have a model of equilibrium that comes from their experience as a child on a see-saw. In fact, a sophisticated view of equilibrium includes fluctuations around a normative equilibrium state ("dynamic equilibrium"). We may show equilibrium atoms exchanging excitement energy.

4 Modeling Tutoring Expertise

A system's evaluation of student behavior, common errors, and plausible misconceptions precedes and informs its generation of appropriate response, be that giving correct answers, elaborating on a student's answer, or providing new information. Enabling a machine to respond appropriately does not mean providing it with rules that enable it to mimic conventional classroom teaching strategies. For instance, in the classroom a teacher might deliberately choose to avoid talking about a student's misconceptions by simply providing the correct answer. This response does not necessarily help the student.

However, misconceptions can be dealt with in our systems. The rich example-based simulations can engage misconceptions as active concepts to be expressed and acknowledged by the learner before learning can take place. These misconceptions can be discussed along with the more formal physics concepts being taught.

One of our goals is to encourage students to entertain the disparities between their conceptualizations and the more formal science as felt conflicts [Claxton, 1985]. Machine responses are geared toward supporting students engaged in the process of conflict resolution. In fact, rather than provide students with the correct answer, an intelligent tutor might increase the student's articulation, exploration, and expression of his alternative conceptions so that the disparity and overlap between the two concepts become clear enough for the student to resolve.

As described in this section, examples, questions, and consequences are used to show students how their conceptions are in conflict with the observable world in terms of descriptions, predictions, actions, and explanations.

Toward this end, we ask experts to provide teaching primitives that can be used to facilitate satisfactory resolution of conflict. How and when to use each primitive is determined by mechanisms based on common teaching strategies [Woolf et al., 1988] such as an example generator, [Woolf & Suthers, submitted] or discourse mechanism [Woolf & Murray, 1987]. A few teaching primitives provided by our experts include:

Questions:

- Qualitative and Quantitative Questions asked by a Student:
  - (e.g., What is the direction of movement of the particle?; What is the mass of an object?).
- Qualitative and Quantitative Questions asked by the System:
  - (Similar to "Questions asked by a Student" above.)
- Question Parameters:
  - (e.g., Identify the variables, math level, relevance of each question).

Examples to Present to Student:

- Easy ("start-up") examples.
- Standard textbook ("reference") examples.
- Counter-examples, strange, hard-to-grasp anomolies.
- General fill-in-the-blanks template-like examples.

Instructional Design:

Provide explicit procedures for explaining topics.
Teach self-diagnosis and self-correction.
Teach descriptive knowledge.

Reasoning about Machine Response. Machine reasoning about construction of response is based on using the primitives listed above. How and when these primitives are invoked is detailed in other work (see Woolf et al. [1988]). However, a few such techniques are outlined here.

If a student's error is suggestive of a deep yet imprecisely identified misconception, the general approach is to teach by demonstrating the consequences of the student's misconception. In such a case, the system might simulate the results of the indicated force lines on the crane boom or demonstrate the effect of a student's suggestions on the equations.

If the specific misconception is known and multiple evidence for it manifested, then the tutor teaches by example. The generated or selected example might be a simple one, an anchor, or an extreme example.

If the misconception is considered to be a simple oversight, such as that the requested force line or answer to a question was handled correctly in previous examples, then the machine teaches by guidance and perhaps provides a hint or the missing rule.

These general rules apply to interactive simulations for any topic. A few specific rules for the statics tutor are outlined below.

- The tutor demonstrates the consequences of the student's incorrect action either by asking him/her to consider the resulting equations (which would show a contradiction); by asking a qualitative question about the balance of force or torque vectors (which could also show a contradiction); or graphically on screen by having the physical system move according to the forces indicated by the student.

- The tutor asks the student about analogous cases, simple cases in which the student's intuitions are valid, or extreme cases in which an unphysical outcome is clear on qualitative grounds. The tutor follows up by asking the student to describe similarities and differences between analogous examples.

- The tutor leads the student through a kind of mental check-list; for example, by asking the student, for each force indicated, what body exerts that force, or by asking the student to list each of the bodies in contact with the boom.

- The tutor asks leading questions or gives a more-or-less direct hint.

- The tutor informs or reminds the student of a relevant fact or principle and asks him/her how that applies in the present case.

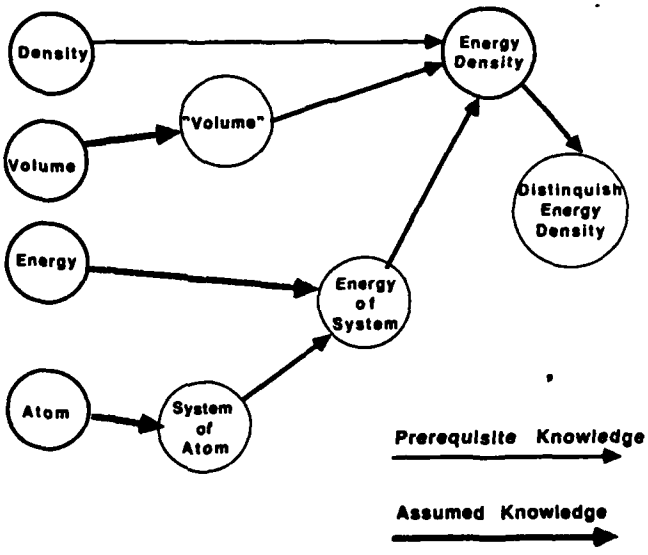


Figure 4: Topic Breakdown for Thermodynamics

- The tutor simply informs the student of the right answer or method and then proceeds.

5 Modeling Reasoning in Science

In this section we highlight the extent to which our experts were able to provide topics and domain knowledge consistent with building intelligent tutors. We asked experts to provide:

Topic Breakdown:

A map of the (sub)topics of the domain.
Concepts and topics assumed known by students.
Type of knowledge for each topic:
    Facts, processes, system, descriptive, or prescriptive knowledge.
    First principles or meta-knowledge.
Relationship between topics:
    Prerequisite, generalization, specialization, or analogy knowledge.

The topics shown in Figure 4 were used to build the knowledge base for the thermodynamics tutor. However, having elicited this knowledge from the expert, we recognized that we needed additional information about relations between topics. Thus, each of the following attributes for each of the topics was requested and represented in the arcs of the semantic network:

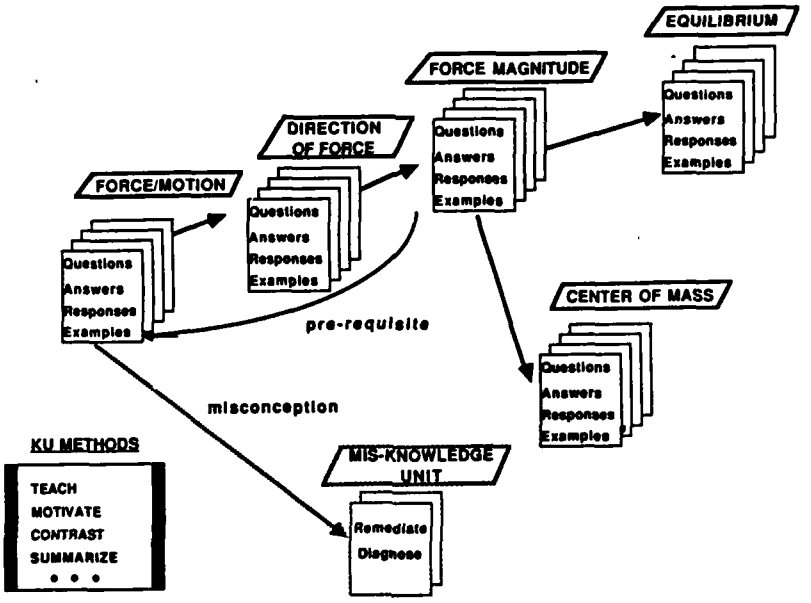


Figure 5: Hierarchy of Frames

importance of the topic
complexity of the topic
prerequisite topics
supports provided for other topics
causes other topics
temporally related to other topics
provides evidence for other topics

Knowledge for both tutors was represented in TUPITS[3], an object-oriented system written on top of a Lisp-based knowledge representation language.[4] TUPITS is an attempt at a systematic representation of the primitive components of tutorial discourse interaction and the rules or strategies which use these components. The primary objects in the system are:

- Lessons: define high-level goals and constraints for each tutoring session;
- KUs (knowledge units): corresponds to the smallest grains of information to be taught;
- MIS-KUs: represent common misconceptions, wrong facts or procedures, and other types of "buggy" knowledge;
- Examples: specify such things as example situations, diagrams, and parameters which configure a learning environment or simulation in a particular way;

[3]Tutorial discourse Primitives for Intelligent Tutoring Systems
[4]HPRL, Hewlett-Packard Representation Language.

- Questions: define tasks for the student to engage in and how the student's actions during the task are to be evaluated and responded to; and
- Presentations: (described below).

Examples are defined in a general way so that they can be used in a variety of situations, such as giving an example of something, asking a question about an example situation, comparing it with other examples, etc. Questions are also defined in general terms so that they can apply to an entire class of examples. Presentations define specific student/tutor interaction events, which can be as simple as telling the student some fact but typically specify a pairing of an Example (or learning environment) with a Question (or task for the student to attempt).

As shown in Figure 5, each KU in TUPITS is represented as a frame and each frame is further linked together with other frames that represent prerequisites, co-requisites, or triggered misconceptions.

Each object has a variety of information associated with it that allows the system to respond dynamically to the tutorial situation. For instance, Knowledge Units, or topics represented as objects, know how to (or have "methods" to):

- teach their own topic interactively;
- explain knowledge didactically;
- teach their own prerequisites;
- test students for knowledge of that topic;
- summarize themselves;
- provide examples of their knowledge (an instantiation of a procedure or concept);
- provide motivation for a student learning the topic; and
- compare this knowledge with that of other KnowledgeUnits.

Similarly, Examples and Questions contain information pertaining to various potential uses of the object. Examples know how to introduce themselves and can point to other related Examples, such as extreme cases of themselves. Questions have information concerning hints, challenges, congratulations, etc., specific to the question. Mis-KUs have methods for diagnosing the presence of their mis-knowledge in the student and for attempting to remediate the bug.

Figure 6 gives a flavor of the frame structures used. It is not an actual example of a frame object used by the system, but rather a combination of some important slots from an Example, a Presentation, and a Question.

Representing the various tutorial discourse primitives in an object-oriented system allows us to define methods and default information differently for various classes of

```
        (Description: A system of atoms is ...)
        (Motivation: We would like to be able to ...)
        (Summary: Systems of atoms are ...)
        (Prerequisite: atoms, energy, ...)
        (Presentation: Explain-sys-of-atoms-1
                (purpose: question)
                (setup: put 10x10 random 15%-1 on screen)
                (question:
                    (q-text: How many ON atoms are in the system?)
                    (answers: (0, 15, 50, 80, 85, 100))
                (correct: 15)
                (hints: (presentation-11, presentation-12, ...))
                (challenge: Why isn't the answer 50?)
                (congradulate: That's right ...)
                (explain: Actutually, the correct answer is ...)
                (reactions:
                    85
                        (Remediate misconception-white-atoms-are-on)
                    ); end of reactions
                ); end of question
        ); end of presentaiton
);
```

A Primitive KnowledgeUnit

objects of the same basic type. For instance, KUs are classified as being for factual, conceptual, or procedural information and each class has different methods associated with it. Also, individual KUs can override the default methods such as an idiosynchratic teach-prerequisite method defined for a particular KU.

Each object has a wealth of information, only a small part of which is used in any particular use of the object. The behavior of the system (and the aspects of each object which are invoked) at any given time is a function of the current tutoring strategy and the discourse context (such as what kind of knowledge is being presented, how many wrong answers the student has given lately, etc.).

For instance, during one phase of a hypothetical session, KnowledgeUnits with certain characteristics might present a motivation before teaching themselves. Correct answers could be followed by congratulations with no challenges, and no more than two hints would be given before the student was told the correct answer to a question.

The TUPITS system provides a canonical, yet flexible, representation for the primitive components of a tutorial interaction. It facilitates experimentation and a quick test-evaluate-modify cycle in ITS construction. The functionality for representing and dynamically selecting tutoring strategies exists and is fairly easy for instructional and *domain* experts to use. But much work still needs to be done in acquiring effective and detailed strategies from these experts.

KnowledgeUnits, such as described here, can be available for several applications. For example, the database can use them to respond to simple student questions; the tutor can use them to produce a case-based example or situation, related questions, and triggered misconceptions; and the expert system can use them to solve diagnostic or

design problems. A reflective tutor should also use these primitives as a source of meta-knowledge about topics that enables the tutor to answer questions such as "Why is that important?"

6 Conclusion

In summary, we have identified models for reasoning about tutoring, science, and cognitive processes. In the specific cases described, the models represent knowledge about physics concepts (e.g., mass, distance, and force), physics procedures (e.g., equilibrium, torque, and entropy), and problem solving heuristics (e.g., inspect, predict, and manipulate). We described a framework for practical acquisition of such knowledge, having identified generic and consistent tools that represent knowledge and control structures for several of the models. A coordinated AI modeling methodology has shown the models to be explicit, reusable, and generative.

The paper provides a structural description of the interviewing process, based on our need to listen to how domain and teaching experts actually help students understand difficult concepts. The framework constrains both the knowledge engineering and acquisition processes and details the structural design of certain components. This work contributes to the development of authoring systems for intelligent tutoring systems.

References

[Anderson, 1981] Anderson, J., Tuning of Search of the Problem Space for Geometry Proofs, *International Joint Conference on Artificial Intelligence*, British Colombia, 1981.

[Atkins, 1982] Atkins, T., *The Second Law*, Freedman Publishers, San Francisco, CA, Scientific American Series, 1982.

[Clancey, 1987] Clancey, W., Assessment of Designs in Progress, from a talk delivered at the ESE Consortium Meeting on Knowledge Engineering and System Architecture, Hawaii, Nov 1987.

[Claxton, 1985] Claxton, G. L., Teaching and Acquiring Scientific Knowledge, in T. Keen & M. Pope (Eds.), *Kelly in the Classroom: Educational Applications of Personal Construct Psychology*, Cybersystems Inc., Montreal, 1985.

[Clement, 1982] Clement, J., Student's Preconceptions in Introductory Mechanics, *American Journal of Physics*, 50 (1), January 1982.

[Duckworth et al., 1987] Duckworth, E., Kelley, J., and Wilson, S., AI Goes to School, *Academic Computing*, November, 1987.

[Kass & Finin, 1987] Kass, R., and Finin, T., Rules for the Implicit Acquisition of Knowledge about the User, *Proceedings of National Association of Artificial Intelligence*, Seattle, WA, 1987.

[Halff, to appear] Halff, H., Curriculum and Instruction in Automated Tutors, in Polson, and Richardson, (Eds.), *Foundations of Instructional Tutoring Systems*, Erlbaum and Assoc., to appear.

[McDermott, 1984] McDermott, L., Misconceptions in Classical Mechanics, in *Physics Today*, July, 1984.

[Woolf, Murray, Suthers, & Shultz, 1988] Woolf, B., Murray, T., Suthers D., and Schultz, K., Primitive Knowledge Units for Tutoring Systems, in *Proceedings of the International Conference on Intelligent Tutoring Systems (ITS88)*, June, 1988.

[Woolf & Murray, 1987] Woolf, B., and Murray, T., A Framework for Representing Tutorial Discourse, *International Joint Conference on Artificial Intelligence*, Milan, Italy, 1987.

[Woolf & Cunningham, 1987] Woolf, B., and Cunningham, P., Multiple Knowledge Sources in Intelligent Tutoring Systems, in *IEEE Expert*, Summer, 1987.

[Woolf et al., 1986] Woolf, B., Blegen, D., Verloop, A., and Jensen, J., Tutoring a Complex Industrial Process, in *Proceedings of the National Conference on Artificial Intelligence* (AAAI-86), Morgan Kaufmann, Inc., Los Altos, CA, 1986.

Appendix 6-I

Primitive Knowledge Units for Tutoring Systems

Beverly Woolf
Tom Murray
Dan Suthers
Klaus Schultz *

*School of Education
University of Massachusetts
Amherst, Massachusetts 01003

Abstract: We are developing a theoretical framework for practical design of tutoring systems. This framework facilitates knowledge acquisition with those multiple experts involved in the process of building intelligent tutors. The framework includes mechanisms for representing knowledge and control information within the system. These mechanisms are responsible for dynamically customizing the machine's responses. The framework is flexible, domain-independent, and designed to be rebuilt, e.g., decision points and machine actions are intended to be modified through a visual editor. This paper describes the framework and formal reasoning structures now being built. It also provides case studies for how these mechanisms are being applied in the building of intelligent tutors for science education.

This work was supported in part by a grant from the National Science Foundation, Materials Development Research, No. 8751362. Support was also received from Air Force Systems Command, Rome Air Development Center, Griffiss AFB, New York, 13441 and the Air Force Office of Scientific Research, Bolling AFB, DC 20332 under contract No. F30602-85-C-0008. This contract supports the Northeast Artificial Intelligence Consortium (NAIC). Additional support was received from an ONR University Research Initiative Contract No. N00014-86-K-0764.

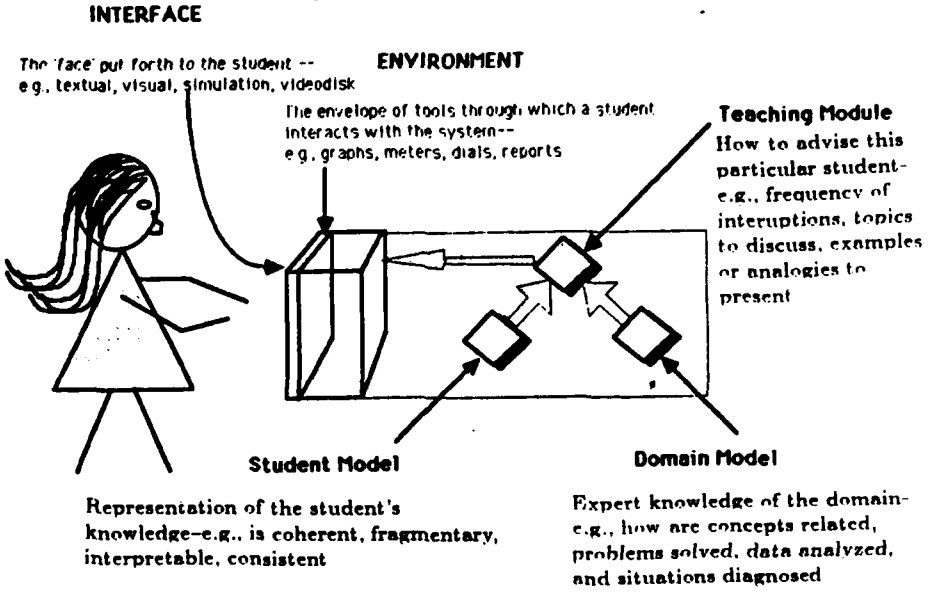


Figure 1: Components of an Intelligent Tutoring System

1 Research Issues

Building an intelligent tutoring system requires the ability to model and reason about domain knowledge, human thinking and learning processes, and the teaching process. Acquiring and encoding this large amount of knowledge is difficult and time consuming. We have been searching for efficient ways to perform these knowledge acquisition and knowledge engineering tasks. This paper describes the process of developing uniform data and control structures that can be used by a wide circle of authors, e.g., psychologists, teachers, curriculum developers, etc., who are involved in the process of building the system [see Woolf and Cunningham, 1987a and 1987b]. One of our goals is to build generic and uniform tools that will enable experts to use the computer directly without the need for a knowledge engineer. To do this, we need to know what knowledge will be supplied by each expert and where that knowledge will be stored.

We are building several tutors in the area of science education to test our understanding of this process. In these tutors we provide students with simplified models of the real world within which they can interact to test their intuitions about physical laws. We would like these simplified models to evolve into simulated laboratories accompanied by mechanical partners or advisors that customize the system's response to the idiosyncrasies of the student.

This paper describes the initial knowledge representations and control structures used in building the prototype tutors. Each system has a model of teaching, domain, and student knowledge and each model contains a representation of knowledge and a control structure that uses, selects from, makes inferences about, and updates that knowledge.

Our research focuses on developing knowledge and control primitives for each module

listed above. In addition, we have been looking at how to encode cognitive processes into primitive structures for the system.

Knowledge Units supply the component elements for tutoring, student, and domain modules. They express the topic to be taught, the possible tutoring responses, and the student's knowledge. Such Knowledge Units should be consistent through out the system and should express the module's knowledge and reasoning. A prototype knowledge representation system for the tutorial components of an intelligent tutoring system is described in Section 3.1.

Control Structures are used to guide movement of the system through the Knowledge Units for each module. The current control structures are tuned for "directive" instruction; that is, they are motivated by specific instructional and diagnostic goals, thus producing a predominantly Socratic interaction. The control structures specify three levels of control, separately defining the topic, presentation, and response selection. We also describe a knowledge representation language that facilitates representing and modifying common tutorial action patterns (Section 4.2). A second tool facilitates example selection (Section 4.2).

Cognitive Considerations includes the knowledge of teaching and learning that enables a system to have expectations and make inferences about a student's actions. Additionally, the more information the system has about typical student errors and potential student misconceptions, the more it can base its decisions on tutoring experience and expertise. Our tutors do not yet represent these cognitive considerations. We have to complete our understanding of teaching and learning in each domain in order to guide machine inferences about student behavior.

In addition to completing our understanding about knowledge, control, and cognitive considerations, we need to make these primitives accessible to the human teachers who might modify them through on-line visual editors. Such visual editors will allow a teacher to use graphic representations, similar to those in Figure 8, to reconfigure the machine's control knowledge.

2 Case Examples: Building Tutors For Statics and Thermodynamics

We have built two tutors in conjunction with the Exploring Systems Earth (ESE) Consortium [Duckworth et al., 1987].[1] These tutors are based on interactive simulations that

[1] *ESE is a group of universities and industries working together to build intelligent science tutors.* The schools include the University of Massachusetts, San Francisco State University, and the University of Hawaii, supported by the Hewlett-Packard Corporation.

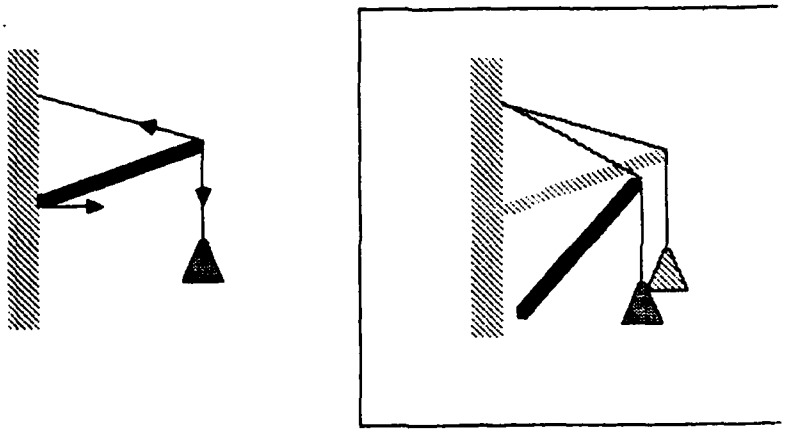


Figure 2: Statics Tutor: Simulation

encourage students to work with "elements" of physics, such as mass, acceleration, and force. The goal is to help students generate hypotheses as necessary precursors to expanding their own intuitions. We want the simulations to encourage them to "listen to" their own scientific intuition and to make their own model of the physical world before an *encoded tutor advises them* about the accuracy of their choices. Identifying and encoding the cognitive considerations listed in Section 1 begins to allow us to track the student's cognitive processes as she develops such hypotheses. These tutors have been described elsewhere (see for example, Woolf & Cunningham, 1987a; Woolf & Murray, 1987) and will only be summarized here.

Figure 2 shows a simulation for teaching concepts in introductory statics. In this example, students are asked to identify forces and torques on the crane boom, or horizontal bar, and to use rubber banding to draw appropriate force vectors directly on the screen. When the beam is in static equilibrium there will be no net force or torque on any part of it. Students are asked to solve both qualitative and quantitative word problems.

If a student were to specify the forces indicated by the heavy vectors in Figure 2A, the solution would be incomplete: a force vector component is missing (located at the wall and pointing upwards).

There are many possible responses to such a student error, depending on the tutorial strategy in effect. The tutor might present an explanation, a hint, provide another problem, or demonstrate that the student's analysis leads to a logical contradiction. Still another response would be to withold explicit feedback concerning the quality of the student's answer, and to instead demonstrate the consequence of omitting the "missing" force: i.e., the end of the beam next to the wall would crash down. Such a response would show the student how her conceptions might be in conflict with the observable world and

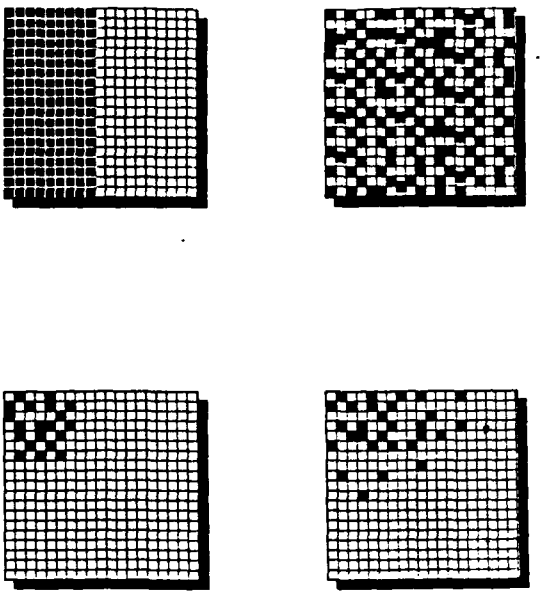


Figure 3: Thermodynamics Tutor: Simulation

to help her visualize both her internal conceptualization and the science theory. This last response is indicated in the example in Figure 8 (Section 4.2).

A second tutor is designed to improve a student's intuition about concepts such as energy, energy density, entropy, and equilibrium in thermodynamics. It makes use of a very simplified but instructive simulated world consisting of a two-dimensional array of identical atoms (Figure 3 [Atkins, 1982]).

Like the statics tutor, the thermodynamics tutor monitors and advises students about their activities and provides examples, analogies, or explanations. In this simplified world the atoms have only one excited state; the excitation energy is transferred to neighboring atoms through random "collisions." Students can specify initial conditions, such as which atoms will be excited and which will remain in the ground state. They can observe the exchange of excitation energy between atoms and can monitor, via graphs and meters, the flow of energy from one part of the system to another as the system moves toward equilibrium. In this way, several systems can be constructed, each with specific areas of excitation. For each system, regions can be defined, and physical qualities, such as energy density, or entropy, plotted as functions of time.

3 Knowledge Units

Both the statics and thermodynamics tutors were built using Knowledge Units (KUs) to define the knowledge taught by the tutor. KU objects in the system represent topics to discuss, concepts, skills, or facts to be taught, or misconceptions to be remediated. Presently these topics are represented within a Knowledge Unit network of topic frames which explicitly expresses relationships between topics such as prerequisites, co-requisites,

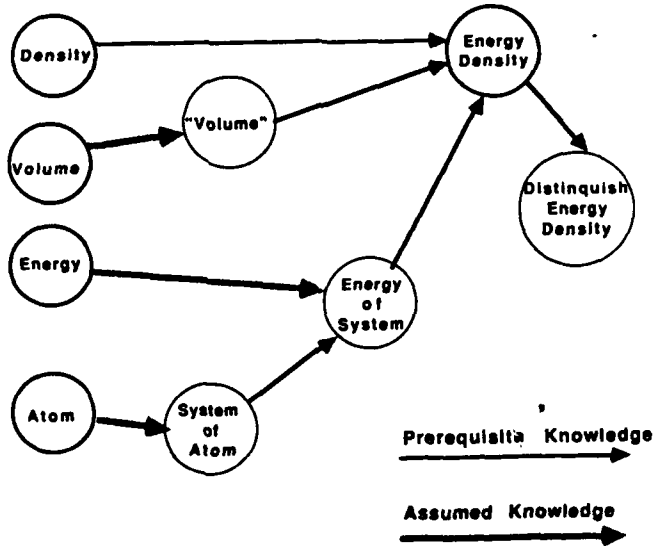


Figure 4: Topic Breakdown for Thermodynamics

and related misconceptions. The network is declarative; it defines a structured space of concepts, but does not mandate any particular order for traversal of this space.

The system's tutoring strategies manifest themselves by parameterizing the algorithm used to traverse the Knowledge Unit network. Several strategies have thus far been implemented. The tutor might always teach prerequisites before teaching the goal topic. Alternatively, the tutor might provide a diagnostic probe to see if the student knows a topic. If the student doesn't exhibit enough knowledge on the probe, then prerequisites might be presented. These prerequisites may be reached in various ways, such as depth-first and breadth-first traversal. An intermediate strategy which we have used is to specialize the prerequisite relation into "hard" prerequisites, which are always covered before the goal topic, and "soft" prerequisites, taught only when the student displays a deficiency.

There are also "Mis-Knowledge Units" (MIS-KUs), which represent common misconceptions or knowledge "bugs" and ways to diagnose and remediate them. These are inserted opportunistically into the discourse. The tutoring strategy parameterizes this aspect of Knowledge Unit selection by indicating whether such remediation should occur as soon as the misconception is suspected or after the current Knowledge Unit has been completed. If enough evidence for a MIS-KU has been collected, then it can be remediated directly. If only partial evidence is available (a "suspected" misconception), then it can be diagnosed and then remediated if confirmed.

The topics shown in Figure 4 were among those used to build the knowledge base for the thermodynamics tutor. The figure shows some information associated with the KUs, and this information is described in detail in the next section.

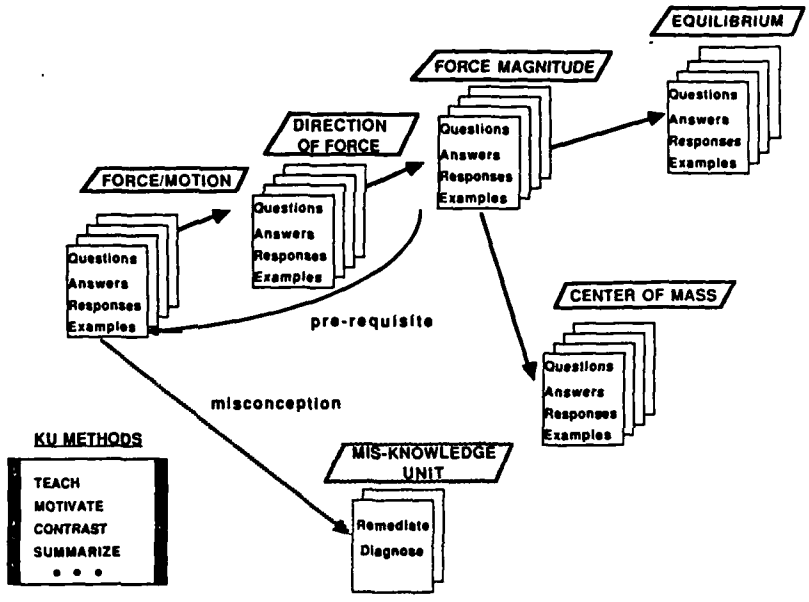


Figure 5: Hierarchy of Frames

3.1 TUPITS

We have been working to define a precise terminology of representational primitives for the following areas: tutorial discourse, examples, types of knowledge, tasks given to the student, student model parameters, and discourse states. Ideally, each of these ontological areas should have a taxonomy of terms that are precise, non-ambiguous, and complete (cover all the possibilities of student action, machine response, etc.).

Toward this end, we have implemented a preliminary system for describing tutorial strategies in terms of a vocabulary of primitive discourse moves. This system, called TUPITS[2], was used to build both the tutors described in Section 2. It is an object-oriented representation language that provides a framework for defining primitive components of a tutorial discourse interaction. These components are then used by the tutor as it reasons about its next action.

As shown in Figure 5, each object in TUPITS is represented as a frame and each frame is further linked together with other frames that represent prerequisites, co-requisites, or triggered misconceptions. Each object has a variety of information associated with it that allows the system to respond dynamically to the tutoring situation. For instance, Knowledge Units, or topics represented as objects, have procedural "methods" associated with them to:

- teach their own topic interactively;
- explain knowledge didactically;

[2]TUPITS (Tutorial discourse Primitives for Intelligent Tutoring Systems) is written on top of HPRL, Hewlett-Packard's Lisp based Representation Language.

- teach their own prerequisites;
- test students for knowledge of that topic;
- summarize themselves;
- provide examples of their knowledge (an instantiation of a procedure or concept);
- provide motivation for a student learning the topic; and
- compare this knowledge with that of other Knowledge Units.

The primary objects in TUPITS (as shown in Figure 5) are:

- Lessons which define high-level goals and constraints for each tutoring session (see Murray, 1987);
- Knowledge Units (KUs) (described in this section);
- MIS-KUs, which represent common misconceptions, wrong facts or procedures, and other types of "buggy" knowledge;
- Examples, which specify parameters that configure an example, diagram, or simulation to be presented to the student;
- Questions, which define tasks for the student and how the student's behavior during the task might be evaluated; and
- Presentations, which bind an example and a question together.

The term "Example" is a generalization of the more typical use of the term and includes giving an exemplar or asking a question about an example situation.

"Question" is also a generalization and is used to refer to any type of task given to the student, such as asking multiple choice questions, solving word problems, and gathering and analyzing data in a simulated laboratory. Questions are also defined in general terms, so that they can apply to an entire class of examples. For example, "Which is larger?" could be used with a number of diverse example situations (each of which, in this case, must be a set of exemplars).

Presentations define specific student/tutor interaction events, which can be as simple as telling the student some fact, but typically specify a pairing of an Example (or learning environment) with a Question (or task for the student to attempt).

Similarly, Examples and Questions contain information pertaining to various potential uses of the object. Examples objects can introduce themselves and can point to other related Examples, such as extreme cases of themselves. Questions have information concerning hints, challenges, congratulations, justifications, etc., specific to the question.

```
(Knowledge unit: SYSTEM OF ATOMS
    (Hard-prerequisites: atoms, energy)
    (Description: A system of atoms is ...)
    (Motivation: We would like to be able to ...)
    (Summary: Systems of atoms are ...)
    (Presentations: pres-16
        (purpose: question)
        (pres-intro: Here is a system of atoms ...)
        (setup: show 10x10 system 15% ON)
        (Question: Q-45
            (q-text: How many ON atoms are in
                                the system?)
            (answers: (0, 15, 50, 80, 85, 100))
            (correct: 15)
            (hints: (presentation-11, presentation-12 ...))
            (challenge: Why isn't the answer 50?)
            (congratulate: That's right ...)
            (explain: Since white represents ON atoms ...)
            (reactions:
                85 (remediate mis-ku-white-are-off))
        ); end of question
    ); end of presentations
);
```

Figure 6: A Primitive Knowledge Unit

Mis-KUs have methods for diagnosing the presence of the the student's mis-knowledge and for attempting to remediate the bug.

Figure 3.1 gives a suggestion of the frame structure used. It is not an actual example of a frame object used by the system, but rather a combination of some important slots from an Example, a Presentation, and a Question.

Representing the various tutorial discourse primitives in an object-oriented system allows us to define methods and default information differently for various classes of objects of the same basic type. For instance, KUs are classified as being for factual, conceptual, or procedural information and each sub-class has a different "describe" method associated with it. Also, individual KUs can override the default methods, such as an idiosynchratic teach-prerequisite method defined for a particular KU.

Each object has a wealth of information, only a small part of which is used in any particular tutorial session. The behavior of the system (and the aspects of each object which are invoked) at any given time is a function of the current tutoring strategy and the discourse context (such as what kind of knowledge is being presented, how many wrong answers the student has given lately, etc.).

For instance, during one phase of a hypothetical session, Knowledge Units with "high importance" might present a motivation before teaching themselves. Correct answers could be followed by congratulations with no challenges, and no more than two hints would be given before the student was told the correct answer to a question. The parameters which specify the tutorial behavior change dynamically.

The TUPITS system provides a canonical yet flexible representation for the prim-

itive components of a tutorial interaction. It facilitates experimentation and a quick test-evaluate-modify cycle in ITS construction. The functionality for representing and dynamically selecting tutoring strategies exists, and it is fairly easy for instructional and domain experts to use. But much work still needs to be done in acquiring effective and detailed strategies from these experts.

4 Control Structures

Our system defines three main decision levels for automated tutoring:

1. Deciding what KU to teach next
2. Deciding what presentations to give next while teaching the current KU
3. Deciding how to respond to the student's behavior for the current presentation

Each of these levels is iterated within the previous one. That is, several KUs may be "taught" or "described" to satisfy an instructional goal; many presentations might be used to teach a KU; and there may be several levels of interacting involved in responding to the student's behavior (such as asking for a justification, giving a hint, re-asking the original question, and congratulating a correct response). The tutor's behavior at each level is determined by tutoring strategies. There are strategies for traversing the topic (KU) network (Level 1); for determining whether a KU has been taught (also Level 1); for deciding whether motivational, diagnostic, etc. presentations will be used prior to "teaching" a topic (Level 2); for selecting the sequence of explanations and example situations (with questions) to present in attempting to teach a topic (also Level 2); and strategies for determining the type and number of "feedback" and "follow-up" presentations used to respond to each of the student's actions (Level 3). "Strategy specialists" continuously monitor and update system parameters to select the appropriate strategies dynamically.

During the past year we have analyzed tutorial discourse to identify some common primitive discourse moves and teaching strategies. See [Murray et al. 87, in press] for a description of one strategy that has been implemented and tested.

Currently our tutors are "directive" in that specific instructional and diagnostic goals motivate the discourse and the flavor of the interaction is Socratic. The simulation environments we are using allow for student experimentation and exploration, but without extensive guidance or interruption (until the student says she's "done"). We intend to extend the knowledge representation and control components to facilitate coaching and student initiated tutorial behavior as well.

4.1 The Dynamic Model

We use the term "dynamic model" to refer to those parts of the tutor's knowledge base that change during the tutoring session. Specifically, the dynamic model consists of

a Student Model, which represents the current state of the student's knowledge (with *reference* to the KUs and MIS-KUs), and a Discourse Model, which can hold information such as the number of correct answers for the current KU and percentage of questions for which hints were given. The dynamic model architecture is similar to a blackboard architecture. A chronological "event queue" records all actions and decisions by the tutor and student. Knowledge Sources called "event analysts" monitor the event queue and post information to the Student and Discourse Models. Event analysts can use the processed information in the dynamic model as well as the "raw" information on the event queue in making their decisions.

4.2 Control tools

We have built two software modules which are invoked in conjunction with the control structure outlined above.

The first component, the "Example Generator" [Suthers & Rissland, submitted] retrieves stored examples and modifies them for presentation to a student. The example generator uses a prioritized list of constraints in generating the appropriate next example to give to the student. For instance, if the student has made many errors, the tutor might present an example that differs only slightly or perhaps only along a single dimension, from the prior example. On the other hand, another situation might require presentation of a more complex example.

Our goal has been to allow the teaching expert to define example selection strategies through general and qualitative rules. We want the machine to respond to directives such as, "When the student is confused, present simpler examples." This is to be contrasted with situation specific rules such as: "If the student answers question # 5 correctly, then present example # 32." The problem with the latter kind of specification is that it is responsive only to local information, i.e., which problem was presented and how the student responded. It ignores situation abstractions derived from a sequence of interactions. It also forces the author into a tedious level of detail.

The strategic component of the example generator is contained in "example generation specialists." Example generation begins by allowing the specialists to post their requests, which are then prioritized as specified by the current tutoring strategy (Figure 7.) This results in a list of requests sorted by priority. Retrieval is done using the list of requests to return the set of examples which satisfies as many of the requests as possible in the order given without letting the retrieved set go empty.

Modification allows us to tailor examples to the particulars of the current situation, filling out an example base of "seed" examples provided by the human expert as required to meet the needs of an individual student. The modifier is given one of the retrieved examples and modifies it as needed to meet any constraints not satisfied by the retrieved example. Constraint propagation and goal protection techniques are used to ensure that no higher property constraint is violated. If the example was modified, the new example is recorded in the example base. This saves processing if similar situations are encountered

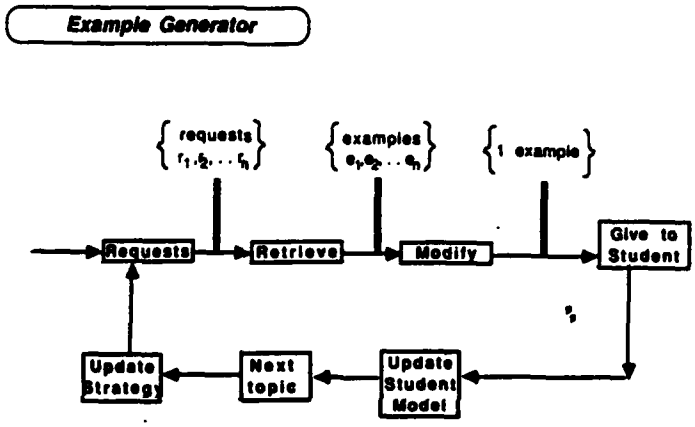


Figure 7: Example Generation Mechanism

in the future. Then the example is returned, to be interpreted by the user interface.

The second control component enables the system to reason about the specification and modification of prototypical patterns of machine action. The graphical nature of this control component allows a user to edit the system's reasoning via a graphical representation. We call this component a discourse action pattern a *Tutoring ACtion Transition Network (TACTN)* [3].

TACTNs facilitate context-dependent constraint of the generation of machine responses [Woolf & Murray, 1987]. They uses a taxonomy of frequently observed discourse sequences to provide default responses for a machine tutor. The TACTN interpreter also uses variables to make choices between alternative discourse sequences. The architecture employs *schemata*, or collections of discourse activities and tutoring responses, as shown in Figure 8, to direct discourse transitions. Some schema are derived from empirical research into tutoring, including studies of teaching and learning [Brown et al., 1986], misconception research [Clement, 1982, 1983], and identification of felicity laws in teaching [van Lehn, 1983].

For instance, if a student shows evidence of misunderstanding an unfamiliar situation, the tutor might replace the current strategy with a Bridging strategy to bridge the conceptual gap from the complex situation to simpler and known ones. A library of such strategies will ultimately be available to the tutor along with reasons why it should move from one strategy to another.

The machine response is generated by traversal through the structure expressed as arcs and nodes. Each schema represents a generalized discourse framework which is invoked when matched to the current student/discourse situation. The action specified

[3]Rhymes with ACT-IN

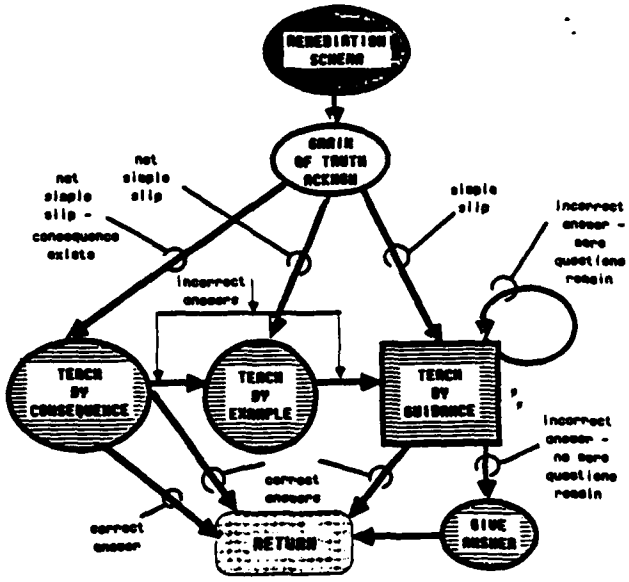


Figure 8: Tutorial Action Transition Network (TACTN)

can be expanded recursively and each is instantiated in a context-dependent way.

Figure 8 represents a simplified set of actions possible with DACTNS. Arcs are defined as predicate sets, which track the state of the discourse, and nodes provide (possibly recursive) actions for the tutor. The situation indicated by the arcs is first assessed, resolving any conflicts between multiply satisfied predicate sets, and then the system carries out the action indicated by the node. This formalism for representing conditional behavior has advantages over generic rule-based systems in that 1) the action is context-dependent, and yet 2) the explicit context need not be incorporated into rule antecedents [Murray 1987].

Discourse control consists of passage through the arcs and nodes of the schemata, with the number and type of schemata depending on context. For example, if the student's answer is incorrect, up to six schemata might be traversed in sequence, including possibly three schemata activated by the Remediation Schemata (Figure 8), Teach by Consequence, Teach by Example, and Teach by Guidance. The exact path through the schemata depends on the tutor's assessment of student error, i.e., whether it was *a simple slip*, *not a simple slip*, or *not a simple slip—consequence exists*.

5 Conclusion

Our goal is to build generic and consistent mechanisms for eliciting and encoding declarative knowledge and control knowledge for an intelligent tutor. We do not have such

generic mechanisms yet, but we are getting closer.

The framework described in this paper, is based on using Knowledge Units and domain-independent control structures. This framework allows us to begin to envision practical construction of tutoring systems for use with experts who are not computer experts. Once developed, such a framework will be used within an authoring system to reduce the excessive time required for building intelligent tutoring systems.

6 References

Atkins, T., *The Second Law*, Freedman, San Francisco, CA, 1982

Brown, D., Clement, J., and Murray, T., "Tutoring Specifications for a Computer Program Which Uses Analogies to Teach Mechanics," *Cognitive Processes Research Group Working Paper*, Department of Physics, University of Massachusetts, Amherst, MA, April 1986.

Clancey, W., "Tutoring Rules for Guiding a Case Method Dialogue," in D. Sleeman and J. S. Brown (Eds.), *Intelligent Tutoring Systems*, Academic Press, Cambridge, MA, 1982.

Clement, J., "Students' Preconceptions in Introductory Mechanics," *American Journal of Physics*, 50 (1), January 1982.

Clement, J., "Students' Alternative Conceptions in Mechanics: A Coherent System of Preconceptions?" *Conference on Students' Misconceptions in Science and Mathematics*, Cornell University, Ithaca, NY, 1983.

Duckworth , E., Kelly, J., & Wilson, S., "AI Goes to School," *Academic Computing*, November 1987.

Littman, D., Pinto, J., and Soloway, E., "An Analysis of Tutorial Reasoning about Programming Bugs," in *Proceedings of the National Association of Artificial Intelligence*, Morgan Kaufmann, Los Altos, CA, 1986.

Michner (Rissland), E., "Understanding Understanding Mathematics," *Cognitive Science*, Vol. 2, 4, 1978.

Murray, T., "Toward a General Architecture for Intelligent Tutoring Systems," Computer and Information Science Technical Report 87-89, University of Massachusetts, 1987.

Murray, T., Schultz, K., Clement., J., Brown, D., "Dealing with Science Misconceptions Using an Analogy-based Computer Program," *The Journal of Artificial Intelligence in Education*, Vol. 1, in press.

Reichman, R., *Making Computers Talk Like You and Me*, MIT Press, 1985.

Stevens, A., Collins, A., and Goldin, S., "Diagnosing Students' Misconceptions in Causal Models," in D. Sleeman and J.S. Brown (Eds.), *Intelligent Tutoring Systems*, Academic

Press, Cambridge, MA, 1982.

Suthers, D. and Rissland, E., "EX Gen: A Constraint Satisfying Example Generator," submitted to *Proceedings of the National Association of Artificial Intelligence* (AAAI-88).

van Lehn, K., "Felicity Conditions for Human Skill Acquisition: Validating an AI Theory," *Report Number CSL-21*, Palo Alto, CA, Xerox Palo Alto Research Center, 1983.

Woolf, B. and Cunningham, P., "Multiple Knowledge Sources in Intelligent Tutoring Systems," in *IEEE Expert*, Summer, 1987a.

Woolf, B. and Cunningham, P., "Building a Community Memory for Intelligent Tutoring Systems," *Proceedings of the National Association of Artificial Intelligence* (AAAI-87), Morgan Kaufmann, Inc., Los Altos, CA, 1987b.

Woolf, B., and Murray, T., "A Framework for Representing Tutorial Discourse," *International Joint Conference on Artificial Intelligence (IJCAI-87)*, Morgan Kaufmann, Inc., Los Altos, CA, 1987.

Woolf, B., and McDonald, D., "Context-dependent Transitions in Tutoring Discourse," *Proceedings of the National Association of Artificial Intelligence*, Morgan Kaufmann, Los Altos, CA, 1984.

Appendix 6-J

Negotiation Among Cooperating Experts

Susan Lander and Victor R. Lesser

1 Introduction

In this paper, we describe a class of distributed problem-solving networks called cooperating experts. Cooperating expert networks have many of the same characteristics as other Distributed Artificial Intelligence (DAI) networks, but have several unique features which require special consideration. We discuss the general problem of coordination in cooperating expert networks and the specific issues that arise from the need to resolve conflicting solutions without a global mediator.

Consider a team of human experts who are cooperating in an office task; for instance, choosing a telephone company. The team consists of a manager and an accountant. They have the shared goal of selecting an appropriate system for the company. However, each individual would like to insure that her own priorities receive top consideration in the overall process. Unfortunately, many of the individuals' local goals and priorities are conflicting from a global viewpoint. For example, the accountant would like to try a company with excellent long distance rates to save money. The manager is concerned about the quality of service, particularly the ability to get connections immediately, and would rather choose a company with a known high level of quality.

In this type of situation, it is very difficult to judge what the "correct" solution is since it depends on the criteria used to decide. In truth, there is no right or wrong answer: the company may need the level of service provided by the second company, but it may also need to save on costs.

Although this is a trivial example of cooperating experts, it is certainly possible to imagine teams working on much larger problems; for example, complex design problems in engineering or architecture, or situation assessment and monitoring tasks. A feature of cooperating expert networks is that there is no one expert who can arbitrate the conflicts which arise. This is true primarily because of the vast amount of knowledge and expertise that can be represented within a group. No individual has all the expertise needed to make informed judgments on conflicts which may originate from within several different disciplines.

Another feature which suggests having a distributed mechanism for conflict resolution is that no one expert has a complete global view of the evolving solution—each has a detailed view of the part of the problem within its area and possibly some abstract view of the overall design. It is possible for each system to have a unique world representation,

inference mechanism, and control strategy which makes it difficult to share information except through formal protocols.

Given the problem characteristics described above, we propose a protocol for negotiating agreements between experts with conflicting goals. Since correctness is an elusive concept, we strive instead for balance. Each expert involved in a decision has some unique insight into the problem. A solution which is acceptable to all experts, though possibly not ideal for any one, is the best we can hope for when there are no global criteria for evaluation.

In the following section, we discuss two major techniques for negotiation: compromise and integrative bargaining. We then describe a network of cooperating knowledge-based systems working in a design domain and present an example of the negotiation of an acceptable design among two systems with divergent goals. The example illustrates the compromise approach to negotiation. We also discuss future work on integrative bargaining.

2 Compromise versus Integrative Negotiation

Pruitt [4] has identified two types of negotiation which are used by expert human negotiators to seek acceptable solutions. These two approaches can also be applied to networks of knowledge-based systems, although it is necessary to take into account the differences in motivation between humans and computer systems. Much human negotiation involves working around personal greed and fears about loss of status, power, and public image. Cooperation is often forced by circumstances rather than by any real desire to produce a mutually satisfactory solution. In our computer networks, we are able to include cooperation as an integral part of the overall problem-solving strategy. This, in turn, leads to negotiation protocols which are more straightforward than those used by humans. We describe the two major approaches to negotiation below.

The first approach, compromise negotiation, involves having each party concede some of it own requirements in order to satisfy others' requirements. This approach is examined in detail in Section 3 with an example from the kitchen design domain. To offer a brief illustration, consider two executives (X and Y) trying to schedule a meeting. X is free from 10:00 until 12:00, plans to lunch from 12:00 until 1:30, and then has meetings with clients until late in the evening. Y has meetings until 12:30, plans to play basketball with friends until 2:00, and then is free until 3:30. X suggests meeting at 10:00, Y says no, how about 2:00?, and X says no to that. Since their initial proposals were unacceptable, they decide to exchange information about their time constraints. It becomes apparent that they have no mutually scheduled free time, and at least one of them will have to give up something that is already scheduled. Each looks for something he can give up that will satisfy the other person's constraints while causing as little disruption to his

own schedule as possible. The solution depends on their priorities and their ability to be flexible with particular times.

The second approach, integrative negotiation, involves identifying the most important requirements of each party and using these to form an innovative solution. Any secondary requirements are relinquished as needed to build the new solution. To illustrate this idea, consider a negotiation over office space. A space planner for an office building has suggested that some existing office space be converted to a conference room. The accountant feels that this is unnecessary and refuses to allow the change. Upon deeper examination of the situation, it is found that the planner is asking for a conference room because the occupants are unhappy with the amount of space available for informal meetings. The accountant, on the other hand, is concerned about the amount of money that can be generated by renting the space and feels that a conference area won't be lucrative. By isolating the underlying concerns, public meeting space and income considerations, an innovative solution can be proposed. In this case, space could be allocated for a cafeteria which would serve as an informal meeting area and still generate enough income. Neither of the experts get what they asked for at a surface level, but both have their most important concerns satisfied.

Integrative negotiation is an exciting alternative to constraint relaxation techniques for negotiation. We are currently attempting to define the information and algorithms necessary to implement it. An expert must be quite "self-aware" in order to identify the underlying motives behind its demands. It is a difficult problem which we are just beginning to explore.

3 An Example of Compromise Negotiation

This section describes the compromise form of negotiation through an example from the kitchen design domain. Kitchen design is the process of laying out components such as cabinets, appliances, and counters in a designated space. The layout must take into account the need for convenience, functionality, aesthetics, and cost effectiveness. The components are grouped into functional units called work centers. For example, a sink, counter, and dishwasher might be grouped together as a cleanup center. The example we present deals with only a small part of the overall task.

There are two experts involved in this portion of the kitchen design task: the Counter Expert and the Appliance Expert. These two experts have some overlap in their functionality. Both experts take an approximate description of the placement of work centers and components in the kitchen. They then choose specific components for the work centers from a global database and generate a design layout with exact specifications.

The initial description of work centers is developed by other experts who concentrate on generating an overall design with all major centers given a tentative and approximate

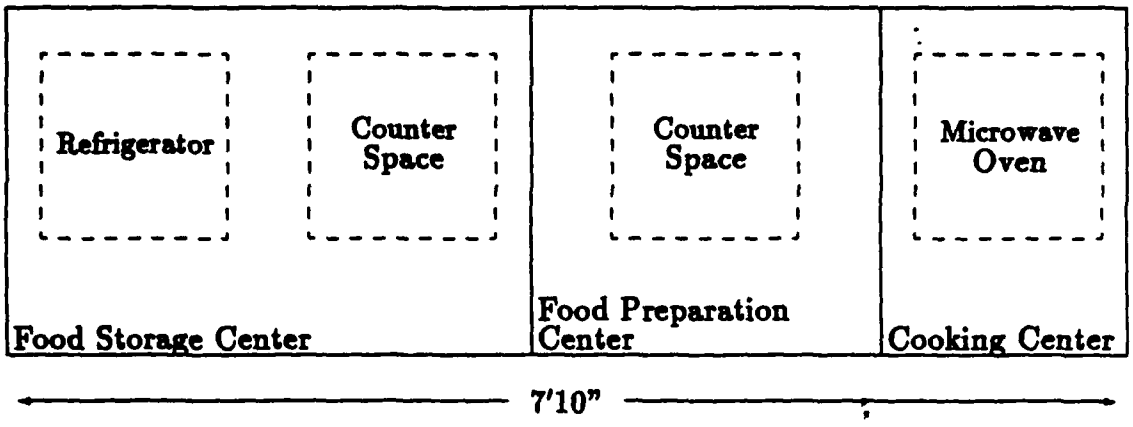


Figure 1: Initial Design

placement. There are three work centers involved in this example: the food storage center, the food preparation center, and a small portion of the cooking center. The food storage center comprises a refrigerator and counter/cabinet space. The food preparation center requires a large open area of counter and accompanying cabinet space for storing needed containers and utensils. The cooking center minimally contains a range (or cooktop and oven) and counter and cabinet space. It may also contain a microwave or convection oven. The only part of the cooking center located in the example region is the microwave oven. We have simplified the example somewhat by focusing on the layout of counter space and appliances only.

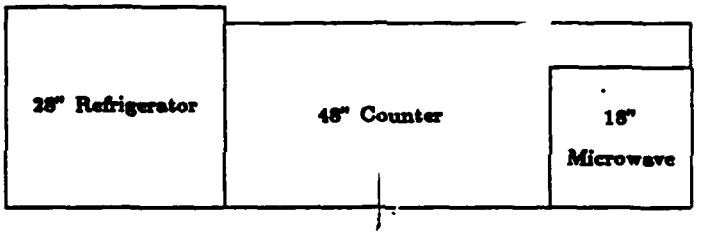
The initial description of work centers is shown in Figure 1. The design is for a region along a wall with a length of 7'10". The food storage center is placed at the left end of the region and the microwave oven is placed at the right end. The food preparation center is to the right of the food storage center.

Figure 2 shows the relevant portion of the component database containing information about available cabinet units and appliances. Microwave ovens range from 18" to 24" widths and refrigerators from 28" to 33" widths.

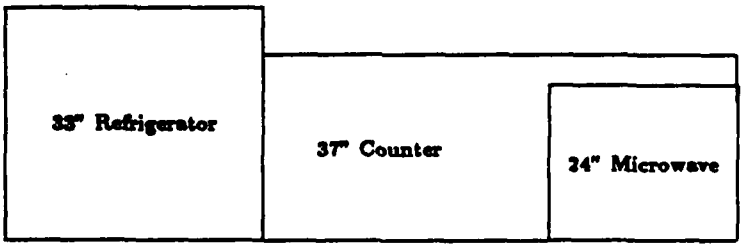
The Counter Layout Expert maintains constraints about how much counter space is needed for each type of work center. The food storage center has a requirement of 18" of counter space adjacent to the refrigerator. The food preparation center has a counter space requirement of 30". These constraints are used by the Counter Expert to generate exact dimensions for counters between appliances. After laying out the counters, the expert applies the heuristic that appliances should be the largest that will fit into the remaining space. Using these constraints and heuristics, the Counter Layout Expert produces Design 1:

REFRIGERATORS	MICROWAVE OVENS
Refrigerator1: width = 33" height = 63" depth = 31"	Microwave1: width = 24" height = 33" depth = 16"
Refrigerator2: width = 31" height = 62" depth = 30"	Microwave2: width = 21" height = 28" depth = 14"
Refrigerator3: width = 28" height = 60" depth = 28"	Microwave3: width = 18" height = 20" depth = 12"

Figure 2: Component Database



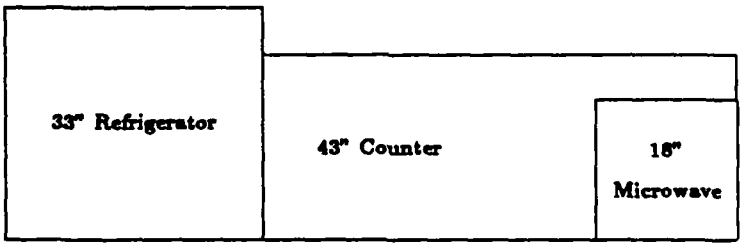
The Appliance Expert has a primary focus of choosing appliances which will provide the greatest functionality. It calculates appropriate appliance size relative to house size and expected occupancy. In our example, the house is 3100 sq.ft. and built for a family of six. Because it is a large home, the appliances chosen are generally the largest available in the database. This expert uses the heuristic that counters fill whatever space is available between appliances and following these guidelines produces Design 2:



Both of these experts have the same initial information to work with and a common high-level goal of producing a detailed design for a region of the kitchen. However the Appliance Selection Expert has a priority of choosing the best appliances and the Counter Layout Expert has the priority of finding enough counter space for the work centers.

Each expert makes its initial design available to other interested parties to evaluate. Design incompatibility is recognized when each of the experts finds its own constraints violated in the other's solution. For example, the Appliance Expert evaluates Design 1 which was produced by the Counter Expert. It evaluates the design in the context of its own constraints; namely, microwave.width= 24″ and refrigerator.width= 33″. Both of these constraints are violated in the design. The Appliance Expert posts a reply to the Counter Layout Expert stating that the design is not satisfactory and including the violated constraints. When the Counter Expert receives this reply it must look for internal constraints which can be relaxed to at least lessen the design's incompatibility with the external constraints that were included in the reply. Each internal constraint has an attribute which indicates whether or not it is relaxable and, if so, indicates some range in which the final values for its variables must lie. Each relaxable constraint also has an attribute which describes how to calculate a new value for the current negotiation iteration.

The experts will continue to iteratively formulate, exchange, and evaluate new designs until acceptable compromise solutions are found for each proposal or until there are no more compromises which can be made. Each acceptable design that is generated will have a rating associated with it that is influenced, at least in part, by the solution's level of compromise. Several acceptable designs are produced from this example. The most highly rated one is shown below. The reason the rating on this solution is high is that much of the compromise is on the size of the microwave oven. The microwave is considered a *non-essential* appliance which makes it less costly to relinquish than either the counter space or the refrigerator.



4 Current Status

We are currently implementing the compromise negotiation strategy described above. We are also attempting to isolate types of information and to develop strategies for the implementation of the integrative negotiation strategy. We can then begin to explore the effectiveness of the different forms of negotiation under specific conditions. Some of the factors which may influence the appropriateness of a particular technique are cooperation and communication strategies, the amount of consideration given to constraints from other systems by an expert, and the amount and type of information available to an

expert through long-term and short-term memory and from other experts. We will be looking at issues such as oscillation and stopping criteria as well.

References

[1] Susan E. Conry, Robert A. Meyer, and Victor R. Lesser. *Multistage Negotiation in Distributed Planning.* Technical Report 86-67, Department of Computer and Information Science, University of Massachusetts, Amherst, Massachusetts 01003, December 1986.

[2] Carl Hewitt. "Offices are open systems." *ACM Transactions on Office Information Systems*, 4(3):271–287, July 1986.

[3] William A. Kornfeld and Carl E. Hewitt. "The scientific community metaphor." *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-11(1):24–33, January 1981.

[4] Dean G. Pruitt. *Negotiation Behavior.* Academic Press, 1981.

[5] Arvind Sathi, Thomas E. Morton, and Steven F. Roth. "Callisto: an intelligent project management system." *AI Magazine*, 7(5):34–52, Winter 1986.

[6] Reid G. Smith and Randall Davis. "Negotiation as a metaphor for distributed problem solving." *Artificial Intelligence*, 20:63–109, 1983.